# Resumption of Runtime Verification Monitors: Method, Approach and Application

Christian Drabek, Gereon Weiss

Fraunhofer ESK
Munich, Germany
e-mails: {christian.drabek,gereon.weiss}@esk.fraunhofer.de

Bernhard Bauer

Department of Computer Science
University of Augsburg, Germany
e-mail: bauer@informatik.uni-augsburg.de

*Abstract*—**Runtime verification checks if the behavior of a system under observation in a certain run satisfies a given correctness property. While a positive description of the system's behavior is often available from specification, it contains no information for the monitor how it should continue in case the system deviates from this behavior. If the monitor does not resume its operation in the right way, test coverage will be unnecessarily low or further observations are misclassified. To close this gap, we present a new method for extending state-based runtime monitors in an automated way, called resumption. Therefore, this paper examines how runtime verification monitors based on a positive behavior description can be resumed to find all detectable deviations instead of reporting only invalid traces. Moreover, we examine when resumption can be applied successfully and we present alternative resumption algorithms. Using an evaluation framework, their precision and recall for detecting different kinds of deviations are compared. While the algorithm seeking expected behavior for resumption works very well in all evaluated cases, the framework can also be used to find the best suited resumption extension for a specific application scenario. Further, two real world application scenarios are introduced where resumption has been successfully applied.**

***Keywords–resumption; runtime verification; monitor; state machine; current state uncertainty; networked embedded systems; model-based.***

## I. Introduction

This paper extends, updates, and provides more detail on earlier research results presented at the International Conference on Trends and Advances in Software Engineering [1].

In various application areas, new kinds of services are created by combining a multitude of different software functions. Off-the-shelf products provide means to connect software functions on a physical and logical level, regardless if the functions are spread over several devices or share a common platform. However, verifying the correct functionality and identifying deviating services remains a challenge, since not only static interfaces have to be compatible but also the interaction behavior [2]. Moreover, the verification process of the final product remains incomplete, as the entire verification of embedded programs is unsolvable in general [3]. Thus, diverse approaches suggest monitoring such a system at runtime to check that it adheres to its specification [4][5][6].

A robust system continues its work after a non-critical failure or deviation from its specification occurs. Hence, it may deviate multiple times during a single execution and all deviations should be identified by the monitor. By this, the development time and effort needed to observe deviations can be reduced; especially, if they are rare and hard to be reproduced. The effort for creating such a monitor can be reduced, if artifacts from the specification phase can be reused [7]. State machines are a common way to specify interactions and protocols. However, these state machines are often limited to expected behavior and have an incomplete transition function. This means, it remains undefined what happens if an unpredicted deviation in the interaction behavior occurs. If the monitor has to terminate on a deviation, any further deviation that would be observable will be missed.

This work presents a novel approach for detecting all differences between an execution of a system and its specification using a single runtime verification monitor. Our main research goal is to enable a monitor to identify all detectable deviations instead of reporting only invalid traces. Moreover, we strive for eliminating the need to split a specification into independent properties. Therefore, we examine how the same monitor instance can resume its observation and find multiple deviations. We call this approach *resumption*.

Using resumption, the same model can be used to define valid behavior in the specification and to verify its implementation, i.e., no separate verification model needs to be created. If available, we suggest to use a reference model of a specification as basis for the monitor. Thereby, it is easier to understand deviations, as they can be directly related to the context of the whole specification. Further, the reuse of the specification guarantees compliance of the respective monitor. We examine the conditions that allow deviations to be identified and the current state uncertainty to be reduced, i.e., when resumption can succeed. We present alternative resumption algorithms and the evaluation framework used to compare them. By selecting a different resumption algorithm, the monitor can be optimized for a particular application scenario. We introduce two real world application scenarios where resumption has been successfully applied.

The rest of this paper is structured as follows. Section II introduces runtime verification using a specification-based monitor and the problem of detecting all deviations. Section III gives a survey of state-of-the-art methods to detect multiple deviations in a system's execution. The method of resumption is introduced in Section IV: First we examine unexpected behavior, before we discuss the detection of deviations and introduce the algorithms considered in this paper. Section V presents the evaluation and discusses the findings. In Section VI, we demonstrate our approach with real world applications. Section VII concludes the paper and gives an outlook on future work.

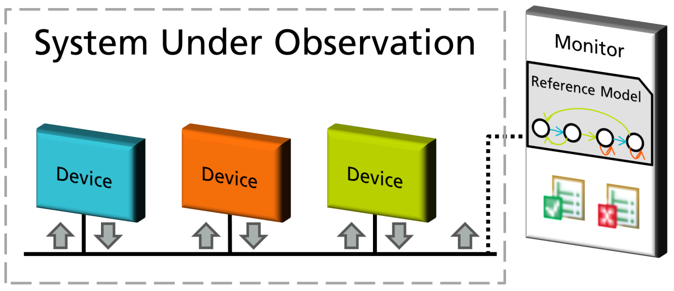


Figure 1. Monitor using a reference model to verify communication behavior of a system under observation.

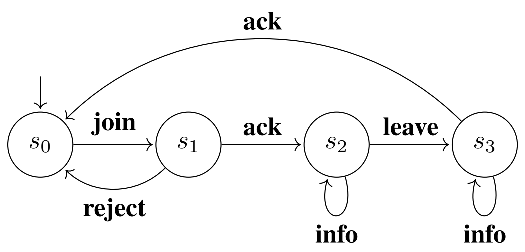


Figure 2. Example illustrating a state machine used to describe the valid communication behavior of a subscription service.

## II. Problem description

This work considers the problem of finding all differences between an execution of a *system under observation* (SUO) and its specification using a single monitor. The core of a monitor is an analyzer which is created from the requirements. Different languages can be used to specify the analyzer [6]. In the literature, several approaches can be found, e.g., linear temporal logic [8]. Such a description can also be given as a set of states and a set of transitions between the states [9], i.e., a (finite) state machine. Further, in automotive and other embedded system domains, state machines are often used for specifying communication protocols or component interactions. We suggest using them in the form of so-called *reference models* [7], which focus on capturing valid behavior and include only critical or exemplary deviations. Such reference models can be learned from observed behavior or generated from other specification artifacts and are quite versatile. They can be used as reference for development, but may also serve as basis for a restbus simulation, or the generation of test cases. Further, a passive reference model can be used as a monitor [7]. Any reference model can be passivated by transforming actions into triggers and introducing intermediate states. It is run in parallel to the SUO and cross-checks the observed interactions with its own modeled transitions (cf. Figure 1). The communication can use a hardware bus, separate links, a middleware or other means. However, we assume the monitor taps into a (virtual) communication bus at a single point and observes the messages in order. Otherwise, this may require additional efforts, e.g., to synchronize times and merge traces, which is beyond the scope of this work. For concurrent behavior, all possible orders are expected to be modeled. The monitor uses the reference model to check the communication and produces verdicts accordingly. As this model is directly derived from the specification, the monitor effectively compares observations with the specification.

A reference-model consists of three main layers: *structual interface*, *mapping to events* and *behavior description*. Structural interface specifications define available messages and their parameters. A mapping defines constraints on the parameters. Thereby, each message can be labeled with a semantic event. The semantic event also captures the sender and receiver of the message. The set of semantic events is used to distinguish the different interactions of the SUO relevant for the specification. At runtime, there are various ways to extract the semantic events by preprocessing and slicing the observed interactions, e.g., [9][7][10][11]. In the following, we will refer to them in general as *events*.

They are used to specify the expected behavior of the SUO as a state machine $\mathrm{SM} = \langle \mathbb{E}, \mathbb{S}, \delta \rangle$.

- $\mathbb{E}$ is the set of events that can be observed.
- $\mathbb{S}$ is the set of states of the state machine, including the initial state $s_0$.
- $\delta \subseteq \mathbb{S} \times \mathbb{E} \to \mathbb{S}$ is the relation of transitions. It is incomplete, as unexpected behavior is omitted.
- Its size is defined as $|\mathrm{SM}| = |\mathbb{S}| + |\delta|$.

An example of such a state machine is shown in Figure 2. Sender and receiver of the events have been omitted for clarity. Where applicable, we use $e_i$ and $s_i$ to refer to events and states as known by the state machine; respectively, $a_i$ and $q_i$ are used for events and states in the sequence of execution by the SUO. For brevity, the labeling function $e(a_i)$ is omitted and $a_i$ is used directly. Therefore, a trace is a sequence of events $a_1 a_2 \ldots a_n = v \in \mathbb{E}^*$. Moreover, $\delta$ is extended to accept traces (1) and sets of states (2).

$$\delta(q_1, v) = \delta(\delta(q_1, a_1 \ldots a_{n-1}), a_n) = \delta(q_n, a_n) = q_{n+1} \quad (1)$$

$$\delta(Q, v) = \{s \in \mathbb{S} \mid \exists q \in Q : \delta(q, v) \mapsto s\} \quad (2)$$

To facilitate referral to events and states with a defined mapping in $\delta$, let $\mathrm{dom}(\delta)$ be the domain of the partial function $\delta$, i.e., the set of elements with a defined mapping. Further, $\mathbb{E}^s$ is the set of events with a defined transition in state $s$ (3) and $\mathbb{S}^e$ is the set of states with a defined transition for event $e$ (4).

$$\mathbb{E}^s = \{e \in \mathbb{E} \mid \langle e, s \rangle \in \mathrm{dom}(\delta)\} \quad (3)$$

$$\mathbb{S}^e = \{s \in \mathbb{S} \mid \langle e, s \rangle \in \mathrm{dom}(\delta)\} \quad (4)$$

A trace $w$ is *valid*, if it describes a path through the state machine SM starting at the initial state $s_0$. Any subsequence of a valid trace is *expected behavior*, i.e., at least one prefix $v_p$ and suffix $v_s$ exist, such that $v_p v v_s = w$. In other words, expected behavior $v$ is contained in a valid trace $w$ and, therefore, $v$ is a sequence of events that can be observed by following a path in the state machine. If the behavior is *unexpected*, it contains at least one event $a_j$ with $\langle q_j, a_j \rangle \notin \mathrm{dom}(\delta)$. An *invalid* trace $\bar{w}$ contains unexpected behaviors and, thereby, contradicts the specification. The SUO *conforms* to the specification if it is in $s \in \mathbb{S}$ and emits $e \in \mathbb{E}^s$. Otherwise, it *deviates* and violates the specification. In the example from Figure 2, any sequence of events obtainable by starting at $s_0$ and following a path through the machine is a valid trace, e.g., *join,ack,info,info,leave,ack*. Any subsequence of this is expected behavior. An invalid trace has no path through the machine, e.g., if *info* is appended to the previous example, the trace becomes invalid and any subsequence containing *leave,ack,info* is unexpected behavior.

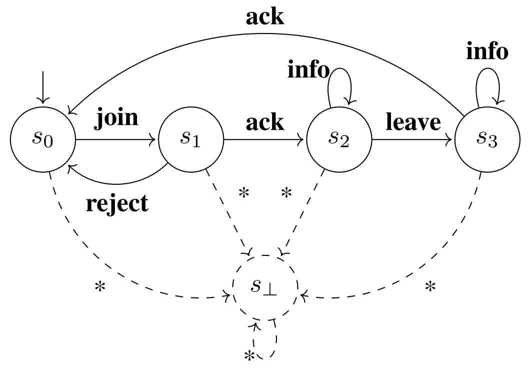


Figure 3. State machine from Figure 2 extended with transitions added for monitoring of any violation without resumption.

A monitor M $= \langle \mathbb{E}, \mathbb{S}, \delta, \mathbb{D}, \gamma \rangle$ defines specific outputs, i.e., the verdicts, for each transition of the state machine that reflect if a violation was detected. In addition to the elements of a state machine, it contains

- a verdict domain $\mathbb{D}$,
- and a verdict function $\gamma : \mathbb{S} \times \mathbb{E} \to \mathbb{D}$.

Usually, it reports only one of the following violation kinds: *invalid traces*, *unexpected behaviors* or *deviations*. Therefore, the verdict domain contains at least an accepting verdict ($\top$) and a rejecting verdict ($\bot$). To produce a verdict for a set of states $Q$, we assume that verdicts can be combined by an operation $\oplus$. This results in the extended verdict function given by (5).

$$\gamma(Q, a_i) = \bigoplus_{q \in Q} \gamma(q, a_i) \tag{5}$$

We call a monitor that uses $\mathbb{E}$, $\mathbb{S}$ and $\delta$ of the reference model a *specification-based monitor*. Because only expected behavior is given in the reference model, each transition corresponds to an expected observation. However, if unexpected behavior is observed, the current state of the SUO becomes undefined by definition. If the deviation was not critical and the SUO is implemented in a resilient way, the SUO will often be able to continue. The SM in Figure 2 shows a SM that specifies the communication behavior related to a subscription service. It has only accepting transitions; a possible resolution of implicit transitions is shown in Figure 3. The wild-card '*' matches any unbound event. Exactly all violations are unbound in the original state machine. Hence, the extended state machine enters the rejecting state $s_\perp$ after a violation. However, as the state machine remains there, the respective monitor only detects invalid traces by a first deviation. To detect further deviations or unexpected behaviors, the monitor must be able to resume its observation.

## III. Related Work

Various areas address the problem of detecting differences between a system's behavior and its specification model. This section gives a brief overview of how existing approaches match specified model and observed behavior.

*Conformance checking* compares an existing process model with event logs of the same process to uncover where the real process deviates from the modeled process [12]. It is used offline, i.e., after the SUO finished its execution, because the employed data mining techniques to match model and execution are computationally intensive. They can only be used efficiently once the complete logs are available. Cook et al. [13] use a best-first search to find the necessary insertions and deletions of events to transform the given event stream into one that exactly matches the model. Based on the *least changes* needed, they give a measure for the difference of an observed trace from its process description. Reger [14] suggests including the origin of an event into the analysis to find sensible edits that are consistent for the same origin and correct the trace. He transforms trace, edits and state machine into weighted transducers $T_T$, $T_X$ and $T_S$. Transducers are composable algorithmic transformations, i.e., automations that read an input sequence and write the result of the transformation as an output sequence. Each step of a weighted transducer is associated with a cost. A three-way composition [15] of the three transducers can be performed in $O(|T_X|\,|T_T|)$ without a large intermediate result. Every path in the composition represents a way to edit the trace to match a path in the state machine. This composed transducer can then be searched for the lowest overall cost edit. The computational complexity to find this edit sequence requires it to be executed offline. In contrast, resumption does not yield precise edits, but can be performed online, in parallel to the execution of the SUO.

Allauzen and Mohri [16] have shown that such a composed transducer can be computed in linear-space. However, the computation is still expensive with regard to time. Therefore, Cook and Wolf [17] utilize *pruning* to reduce the cost of finding a low-cost goal by discarding portions of the search space that look unpromising. They reason that even though pruning breaks the guarantee of identifying the lowest-cost goal, it often has negligible effects on the result while reducing the search space dramatically. They suggest using *cost pruning* and *position pruning*. The former eliminates nodes that are estimated to result in a higher than expected overall cost; the latter removes nodes that are $\tau_{\text{prune}}$ steps in the trace behind the current best node from the search. In contrast to these mining algorithms, the verification with resumption presented in this paper works at runtime. However, we would like to note that, if position pruning is used, all information to perform the search for an edit path is available at runtime with an offset of $\tau_{\text{prune}}$ steps. Therefore, we chose the *least changes* approach for comparison with resumption.

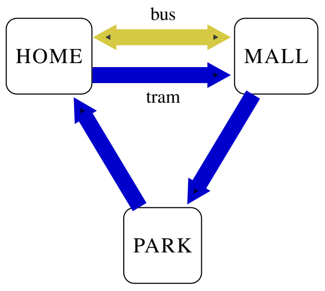
*Model-based testing* aims to find differences between the behavior of a system under test (SUT) and a valid behavior model [18]. An environmental [19] or embedded [20] test context stimulates the SUT with test sets, i.e., selected input sequences. The SUT's outputs are then compared with the expected output from the behavior model. *Homing sequences* actively maneuver the SUT into a known state [21]. Generally, these sequences reduce the current state uncertainty by utilizing separating or merging sequences [22]. Former assure different outputs for two states, latter move the machine into the same state for a given set of initial states. Minimized Mealy Machines are guaranteed to have a homing sequence [22]. However, a passive monitor cannot influence the SUO. Therefore, it cannot actively force the system to a known state. Nevertheless, we utilize occurrences of separating and merging sequences to reduce the number of possible candidates for the current state.

In general, *runtime verification* can be seen as a form of passive testing with a monitor, which checks if a certain run of a SUO satisfies or violates a correctness property [5]. Such a dynamic analysis is often incomplete, i.e., it may yield false negatives; however, this incompleteness helps neutralize limitations of static analysis [9], e.g., state-explosion. The observation of communication is well suited for black box systems, as no details about the inner states of the SUO are needed. Further, the influence on the SUO by the test system is reduced by limiting the intrusion to observation. In case the deviations are solely gaps in the observation, a Hidden Markov Model can be used to perform runtime verification with state estimation [23]. Runtime verification frameworks, such as TRACEMATCHES [10] or JAVAMOP [11], preprocess and filter the input before it is passed to a monitor instance. Thereby, each monitor only observes relevant events. These stages implement the first two layers of a reference model [7]. The monitor uses the reference model's third layer to answers *yes* or *no* to the question, if the provided trace fulfills or violates the monitored property. This is also the case, if it contains multiple violations. However, it may be of interest to identify all violations, similar to conformance checking. Simply keeping the monitor running after it encountered and reported a violation only works in very specific scenarios. This is similar to using the resumption algorithm *Waiting* presented in Section IV-D. Nevertheless, if the properties are carefully chosen, multiple instances of the monitor can match different slices of an input trace [10][24]. Such properties can be extracted from the behavior model [3] or by data mining techniques from a running system or traces [17][21][25][26][27]. However, the former implies additional design work and the latter requires a correctly working system to learn from. Furthermore, this creates a secondary specification that needs to be maintained and validated. In contrast, the presented resumption enables reuse of an available specification by automatically augmenting it with robustness for verification.

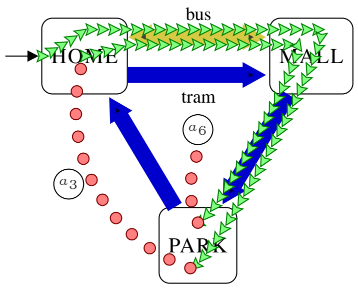
## IV. RESUMPTION

A specification-based monitor, such as shown in Figure 3, will only be able to find the first deviation from a specification, since it enters the final state $s_\perp$ at this point. Different techniques can be applied in order to create robust monitors and to find deviations beyond the first. Up to now, this is usually done manually and requires additional design work, e.g., to repeatedly add transitions and triggers or to artificially split the specification into multiple properties that can be checked separately. Therefore, we suggest using a generic definition for how a monitor can resume its duty. This section describes *resumption*, a method that enables a monitor to analyze a trace for more than one violation with respect to the same property.

For illustration, runtime verification can be seen as a game for two players: DECEIVER and VERIFIER. DECEIVER acts as SUO and he covertly moves on a map, which represents SM, while leaving a trace of moves. After each move, VERIFIER, the monitor, has to tell if DECEIVER violated the specification given by the map. She knows only the trace and the map. Figure 4a shows an example map for the game. Figure 4b depicts the path that DECEIVER has chosen in this example, producing a trace *bus,tram,bus,bus,tram,bus*. He starts at HOME. On step $a_3$ and $a_6$, he choses to *deviate* and claims to move using *bus*, actually not available at state PARK. For $a_3$, he decides to go



(a) Example map known by DECEIVER and VERIFIER.



(b) DECEIVER's hidden movement. Arrows show conforming, circles deviating moves.

Figure 4. Example illustrating a game of runtime verification, where VERIFIER (monitor) tries to detect, if DECEIVER (SUO) violates the specification given as a map (state machine).

HOME; for $a_6$, he may still jump to any state. All other moves *conform* to the map.

Knowing the current state of DECEIVER it is trivial for VERIFIER to follow the conforming movement of DECEIVER using only trace and map. She can infer his next states using the moves recorded in the trace and $\delta$. Further, she can easily identify his first deviation. Then, DECEIVER's move is not contained in $\delta$. This exemplifies how monitors without resumption work.

*Theorem 1:* VERIFIER can detect deviations using the verdict function given in (6), if she knows DECEIVER's current state is $q_i$.

$$\gamma(q_i, a_i) \mapsto \begin{cases} \top, & \text{if } \langle q_i, a_i \rangle \in \mathrm{dom}(\delta) \\ \perp, & \text{otherwise.} \end{cases} \tag{6}$$

*Proof:* If VERIFIER knows DECEIVER is in state $q_i$ and the event is $a_i$, she can look up his next state in $\delta$. If $a_i$ is available at $q_i$, i.e., $\langle q_i, a_i \rangle \in \mathrm{dom}(\delta)$, DECEIVER cannot deviate using $a_i$. If $a_i$ is not available at $q_i$, he has to deviate to be able to use it. Thereby, if VERIFIER knows the current state of DECEIVER, she can directly tell if he deviates. ■

However, after a deviation, VERIFIER does not know to which state DECEIVER has moved. She is uncertain of his current state. Becoming aware of this uncertainty and reducing it in order to be able to resume runtime verification is *resumption*. To express multiple deviations in a sequence of verdicts, each verdict refers to the trace after the last reported violation. First, we show that detection of all unexpected behavior is possible through resumption. Later, we demonstrate how resumption can be employed to find minimal subsequences of a trace, each containing exactly one detectable deviation.

## A. Resumption for Unexpected Behavior Detection

Unexpected behavior is defined as any sequence of events the SUO may not emit according to the specification. Behavior is unexpected if it cannot be matched to anywhere in the specification. We assume it is unknown how the behavior of the SUO's components is implemented internally and only their stimuli and responses are observable (black-box assumption). We expect these interactions in the SUO to be specified by a state machine SM. If the observed events can be matched to SM, such behavior is assumed to conform to the specification. Otherwise, i.e., in case of a violation, the new state of the SUO is unknown as the observations are the only available information. In general, no restrictions on the new state can be given, i.e., such a transition may even be non-deterministic. A monitor M knows only the specification and observes the events emitted by the interactions in the SUO.

More formally, unexpected behavior is a trace that contains an event $a_j$ with $\langle q_j, a_j\rangle \notin \mathrm{dom}(\delta)$. As $q_j$ is unknown to VERIFIER, it is replaced by $Q^k_j = \delta(\mathbb{S}, a_k \dots a_{j-1})$, the set of states reachable with expected behavior from any state with the trace starting at step $k$ and ending before step $j$. Please note that $Q^j_j = \mathbb{S}$.

*Theorem 2:* A trace can be split within space $O(|\mathrm{SM}|)$ and time $O(|\mathbb{S}|)$ per step so that each segment contains exactly one unexpected behavior. For each appended event, it can be decided in this space and time if it belongs to the same or a new segment.

*Proof:* For brevity, we define $\rho_{i+1} = \rho'_i$. A valid segmentation of the trace is a sequence of indexes determined by (7). Each of the segments is delimited by two indexes $\rho_i$ and $\rho'_i$. The trace $v_i = a_{\rho_i} \dots a_{\rho'_i}$ always encloses exactly one unexpected behavior. If there was no unexpected behavior in $v_i$, then $Q^{\rho_i}_{\rho'_i}$ would not be empty. If there was additional unexpected behavior at step $\rho$ with $\rho_i < \rho < \rho'_i$, then $Q^{\rho_i}_{\rho}$ would be empty already and $\rho$ would have been chosen as minimum. Therefore, exactly one unexpected behavior is contained in $v_i$.

$$\rho_0 = 1 \wedge \rho'_i = \min\{\rho \mid \rho > \rho_i \wedge Q^{\rho_i}_{\rho} = \emptyset\} \tag{7}$$

For each step, the set of successors of the possibly active states must be calculated, i.e., $\delta(Q^{\rho_i}_{\rho'_i}, a_i)$. This requires up to $|\mathbb{S}|$ lookups in the transition relation of size $|\delta|$. Each lookup takes time $O(1)$, if perfect hashing is used. Therefore, each step needs time $O(|\mathbb{S}|)$ and space $O(|\mathbb{S}| + |\delta|) = O(|\mathrm{SM}|)$. ■

To relate this to resumption, we explain how VERIFIER checks DECEIVER's movement with no information about DECEIVER's initial state. VERIFIER has to find a verdict for all possible locations of DECEIVER. The transition function $\delta$ is already extended by (2) to accept a set of states $Q$ and to return all states reachable with event $e$ from a state in $Q$. $\delta$ will return an empty set if none of the possible states has a matching transition for the event. To continue verification, this function needs to be extended with the actual resumption. VERIFIER's uncertainty of DECEIVER's state resets if she cannot confirm his movement. The most general assumption is that DECEIVER is in any state. This is reflected in transition function $\delta^+$ (8).

$$\delta^+(Q,e) = \begin{cases} \delta(Q,e), & \text{if } \gamma\text{(Q,e)} = \top \\ S, & \text{otherwise.} \end{cases} \tag{8}$$

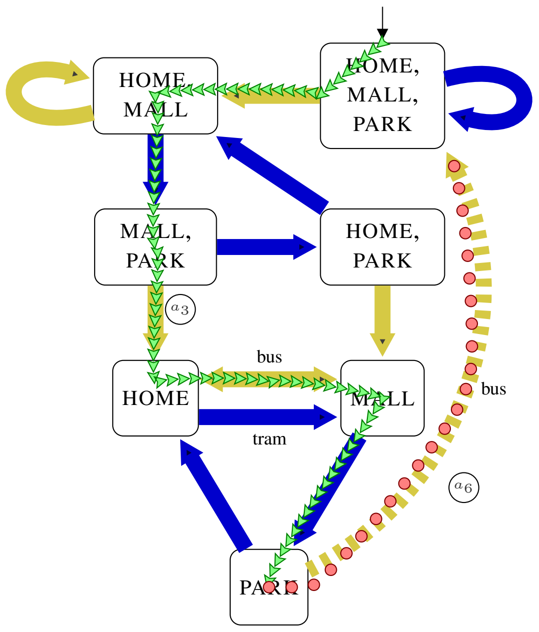


Figure 5. Power map of the example from Figure 4a including the interpretation of DECEIVER's trace for unexpected behavior from VERIFIER's perspective.

Similar, the verdict function $\gamma$ is extended by (5) to produce a verdict for multiple states using a combination operation $\oplus$. For detecting unexpected behavior, this may be done using $\oplus_{eb}$ (9), where $\Gamma$ is the set of verdicts to merge. As long as $Q^e$ is not empty, there is a state in $Q$ with an accepting transition for $e$ and the behavior remains expected. Otherwise, unexpected behavior is detected and the next segment of the trace starts.

$$\oplus_{eb}(\Gamma) = \begin{cases} \top, & \text{if } \top \in \Gamma \\ \bot, & \text{otherwise} \end{cases} \tag{9}$$

Instead of analyzing the model at runtime, power set construction can be used to resolve the non-determinism beforehand. Also, this allows to illustrate resumption for the game example. DECEIVER's moves are translated to the *power map* $\mathrm{SM}^{\mathcal{P}} = \langle \mathbb{E}, \mathcal{P}(\mathbb{S}), \delta^{\mathcal{P}} \rangle$, starting at state $s_{\mathbb{S}}$. $\mathcal{P}(\mathbb{S})$ is the set of all subsets, the power set, of $\mathbb{S}$. $\delta^{\mathcal{P}}$ is $\delta^+$, except that it returns the respective state for the set of states instead of the set itself. Transitions that were unexpected behavior in SM are routed to $s_{\mathbb{S}}$ in $\mathrm{SM}^{\mathcal{P}}$. $\delta^{\mathcal{P}}$ is complete and deterministic. Thereby, VERIFIER always knows DECEIVER's precise state in $\mathrm{SM}^{\mathcal{P}}$ and merely has to follow his movement (see Theorem 1). She starts assuming DECEIVER is in any state of the original map, i.e., in $s_{\mathbb{S}}$ (cf. Figure 5). After step $a_3$, the states DECEIVER could be in using only expected behavior are narrowed down to HOME. As we know from Figure 4b, DECEIVER deviated at $a_3$ to HOME. However, this is still expected behavior as there would be no deviation if DECEIVER started at MALL. This is shown in Figure 6. The second deviation at $a_6$ is unexpected behavior, as there are no other valid options left. Compared to Theorem 2 above, checking for unexpected behavior in $\mathrm{SM}^{\mathcal{P}}$ only takes time $O(1)$ for each step; however, the extended map requires significantly more space: $O(2^{|\mathbb{S}|})$.

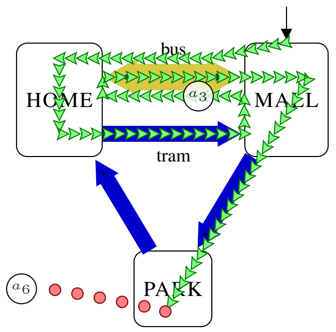


Figure 6. Alternative of DECEIVER's hidden movement producing the same trace as Figure 4b, but with no deviation at $a_3$ by starting at a different state.

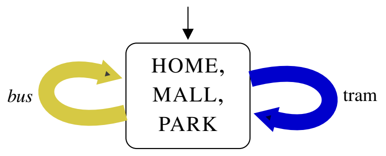


Figure 7. Power map of the example from Figure 4a for deviation detection from VERIFIER's perspective.

## B. *Resumption for Deviation Detection*

The previous section revealed that it is possible for a monitor to detect all unexpected behaviors of a SUO. This section examines how resumption can help to detect deviations of a SUO. A deviation is an event that is not allowed in the SUO's current state. Therefore, deviations are directly linked to the SUO's state. In terms of the example, VERIFIER must detect if DECEIVER chose to deviate with a move. This changes how VERIFIER can handle uncertainty about DECEIVER's state. She may be unable to decide if DECEIVER deviated. This requires a third verdict to express that she is *inconclusive* ($\top\!\!\!\!\perp$).

*Theorem 3:* VERIFIER is exactly then conclusive about DECEIVER's move $a_i$, if the move is defined or undefined for all states $Q$ she knows he could be in (10).

$$\gamma(Q, a_i) = \top\!\!\!\!\perp \iff \exists s_1, s_2 \in Q : \gamma(s_1, a_i) \neq \gamma(s_2, a_i) \tag{10}$$

*Proof:* The implication from left to right follows from the limitations of DECEIVER's moves. If none of the states in $Q$ has a transition for the event, DECEIVER cannot conform; therefore, he *obviously deviates*. If all of the states in $Q$ have a transition for the event, DECEIVER cannot deviate; therefore, he *obviously conforms*. In all other cases, VERIFIER cannot give a conclusive verdict, as there is at least a state $s_1$ in $Q$ that has a transition for event $a_i$ and a state $s_2$ that has not. If her verdict was $\perp$, DECEIVER may in fact have been in $s_1$ and actually conforms; if it was $\top$, he may have been in $s_2$ and deviates. Thereby, her verdict must be $\top\!\!\!\!\perp$. If the verdict is the same for all individual states in $Q$, it must be $\top$ or $\perp$ and it follows the implication from right to left. ■

This helps to formulate $\oplus_d$ (11) for combining verdicts. Identical verdicts combine to the same verdict and different verdicts combine to an inconclusive verdict.

$$\oplus_d(\Gamma) = \begin{cases} \top, & \text{if } \{\top\} \equiv \Gamma \\ \perp, & \text{if } \{\perp\} \equiv \Gamma \\ \top\!\!\!\!\perp, & \text{otherwise} \end{cases} \tag{11}$$

This sounds promising, as at least in some cases there can be conclusive verdicts. Further, one might expect that VERIFIER should be able to close in on DECEIVER's position eventually. However, as it is always possible that DECEIVER deviates, $Q_{i+1}$ must be the set of all states, unless he obviously conforms. This is enforced by $\delta^+$.

As a consequence, VERIFIER can only reduce her current state uncertainty, if she observes a sequence of obviously conforming events. Further, the sequence must eliminate s-tates from her current state uncertainty. In the following, we examine if such sequences exist. Sandberg [22] describes how *synchronization* and *homing* sequences can be found in completely specified, deterministic Mealy machines in time $O(|\mathbb{S}|^3 + |\mathbb{S}|^2 \cdot |\mathbb{E}|)$ — in case one exists. A synchronization sequence is a trace $x \in \mathbb{E}^*$ that transitions the machine into a single known state, regardless the initial state, i.e., $|\delta^+(S, x)| = 1$. In essence, it consists of a sequence of *merging* sequences. A homing sequence may also include *separating* sequences. It is a trace $x \in \mathbb{E}^*$ that guarantees different outputs for different target states (12). While a state machine is not guaranteed to have any synchronization sequences, e.g., the automation in Figure 4a has none, a minimized Mealy machine always has a homing sequence [22].

$$\begin{aligned} &\forall s_1, s_2 \in S : \\ &\qquad \delta(s_1, x) \neq \delta(s_2, x) \implies \gamma(s_1, x) \neq \gamma(s_2, x) \end{aligned} \tag{12}$$

Homing sequences can be mapped to the deviation game. SM contains no output and is incomplete. For former, $\gamma$ could act as replacement and latter can be mitigated by using $\delta^+$. Then, we call them *unique sequences*, as they identify a unique state. However, the pure possibility of DECEIVER deviating negates the benefit of a separating sequence as this resets VERIFIER's search; i.e., she has to consider all possible states again after the possible deviation. As there is no other output available, separating sequences would work only under the assumption that no violations occur during resumption. As SM has no merging sequence, $\text{SM}^{\mathcal{P}}$ for deviation detection could be trimmed to Figure 7. VERIFIER knows DECEIVER confirms using *tram* but this does not tell anything about DECEIVER's location. DECEIVER could be in PARK; therefore, if he uses *bus*, VERIFIER is inconclusive and has to restart at $s_{\mathbb{S}}$. An option would be to resolve undefined transitions differently. However, this would impose additional assumptions on the deviations. Nevertheless, we can use the knowledge about unexpected behavior to conclude on deviations.

*Theorem 4:* Each trace segment $a_{\rho_i} \dots a_{\rho'_i}$ of the split trace contains at least one deviation.

*Proof:* For proof by contradiction, we assume that there is no deviation in the trace segment. If there is no deviation, only conforming moves were used by DECEIVER. Then, there must be at least one path in SM matching the trace, i.e., $\delta(\mathbb{S}, a_{\rho_i} \dots a_{\rho'_i}) \neq \emptyset$. Therefore, the current state uncertainty $Q^{\rho_i}_{\rho'_i}$ should contain at least one state. However, this conflicts with (7), that ensures $Q^{\rho_i}_{\rho'_i} = \emptyset$. Hence, the trace segment must contain at least one deviation. ■

The trace segments span from one unexpected behavior to the next. Somewhere within each segment is a deviation. Yet, we want to locate the deviations as precise as possible. The segments are constructed in a way so that they always end with unexpected behavior. Therefore, they cannot be cut on that side. The following will show that their length can be limited by maximizing the start index while retaining the constraint for the trace segment, i.e., that it contains unexpected behavior.

*Theorem 5:* The trace $a_{\kappa_i} \dots a_{\rho_{i+1}}$ is the closest possible enclosure of a deviation before step $\rho_{i+1}$ that retains the notion of unexpected behavior in the segment, with $\kappa_i$ defined by (13).

$$\kappa_i = \max\{\kappa \mid \rho_i \le \kappa < \rho'_i \wedge Q^{\kappa}_{\rho'_i} = \emptyset\} \tag{13}$$

*Proof:* The proof for existence of a deviation in the trace stays the same as for Theorem 4: $Q^{\kappa}_{\rho'_i}$ is only empty if it contains a deviation. Thus, it remains the proof that there is no shorter segment that could enclose the deviation. If $\kappa_i = \rho'_i - 1$ DECEIVER is obviously deviating with $a_{\rho'_i}$, as no state in $\mathbb{S}$ has a transition for the event. Since there must be at least one event to deviate, there cannot be a shorter sequence in this case. Equation (13) selects the greatest start index $\kappa_i$ that still contains unexpected behavior and thereby a deviation. We cannot reduce the end index as it was already selected to be minimal when starting at $\rho_i$ by (7). Starting at a later index than $\rho_i$ can only increase the current state uncertainty, because the later starting segment must consider at least all paths from the earlier, i.e., $Q^k_j \subseteq Q^{k+1}_j$. Moreover, $Q^{\kappa_i}_{\rho'_i}$ can only be empty if $Q^{\rho_i}_{\rho'_i}$ is empty. Therefore, the closest possible enclosure is the trace $a_{\kappa_i} \dots a_{\rho_{i+1}}$. ■

$\kappa$ can be calculated by searching unexpected behavior using backwards steps, starting from step $\rho'_i$. As following transitions backwards may yield up to $|\mathbb{S}|$ source states for the same event, the time needed for each of the $\rho'_i - \kappa_i \le \rho'_i - \rho_i \le |v|$ steps is increased to $O(|\mathbb{S}|^2)$ compared to what was shown for the forward search in Theorem 2. Preparing the reverse lookup tables will require space $O(|\mathbb{S}|^2 \cdot |\mathbb{E}|)$. It follows that there is at least one deviation located in the intersection of unexpected behavior found forward and backward. Moreover, the existence of $\kappa_i$ implies that each segment $a_{\rho_i} \dots a_{\kappa_i - 1}$ must be expected behavior, even if it actually contains deviations. The deviations perfectly mimic expected behavior and cannot be detected. This can also be seen by the options of DECEIVER in $\mathrm{SM}^{\mathcal{P}}$.

*Theorem 6:* Deviations in the trace segment $a_{\rho_i} \dots a_{\kappa_i - 1}$ cannot be detected.

*Proof:* According to Theorem 5, segment $a_{\kappa_i} \dots a_{\rho'_i}$ contains exactly one unexpected behavior and the unexpected behavior happens at or after step $\kappa_i$. As there is only one unexpected behavior in the complete segment, there cannot be another before step $\kappa_i$. Therefore, any deviation that may have occurred in this part of the segment is observed as expected behavior and cannot be detected. ■

The deviations cannot be detected because there is no possibility to distinguish them from expected behavior with the available information. However, if additional or more detailed observations can be obtained from the system, they may become detectable.

Moreover, if the current state uncertainty is reduced to a single state without the (unobservable) occurrence of a deviation, the detected unexpected behavior is the deviation. Therefore, if given enough time between deviations, monitoring with resumption will identify exactly the deviations.

*Theorem 7:* If $n$ non-overlapping unique sequences are observed without unexpected behavior, the assumed state is the actual state of the SUO, unless $n$ or more undetected deviations occurred. However, unless one unique sequence is an obviously conforming synchronization sequence, there may have been at least $n$ deviations.

*Proof:* For the first part, we show that not all unique sequences can be mislead with $n-1$ deviation. As the unique sequences do not overlap, there remains at least one unique sequence $u$ without a deviation. As $u$ does not contain a deviation, it reveals the actual state of DECEIVER. Theorem 1 guarantees the detection of deviations if the current state is known. Moreover, $u$ must be the last of the unique sequences. Otherwise, the deviation would have been detected and there would be unexpected behavior. With an additional $n$-th deviation, DECEIVER can mislead all unique sequences, therefore, his actual state may remain uncertain. For the second part, we recall how unique sequences are constructed. Each sequence is unique, as it reduces $|Q|$ to 1. Like homing sequences, the reduction can be achieved by merging or separation sequences. A merging sequence uses the structure of the state machine, e.g., if transitions for the same event lead from two states to a single one. During a merging sequence, no deviation can occur by definition. As a synchronization sequence consists of concatenated merging sequence, it is obviously conforming and no deviations could have happened when it is observed. The observation of the synchronization sequence itself guarantees the actual state of DECEIVER is known. Separation sequences, however, work differently. They remove states from the current state uncertainty for which DECEIVER would have to deviate for the chosen event. Hence, there is at least one possibility to deviate in each separation sequence. If none of the observed unique sequences is a synchronization sequence, each contains at least one distinct separation sequence. As there are $n$ unique sequences, there may have been at least $n$ deviations. ■

Detecting multiple unique sequences before detecting unexpected behavior may sound unlikely. Nevertheless, sometimes rare deviations are of interest. In this case, there may be many events between deviations that can be used to resume verification. Protocols such as the subscription service in Figure 2, often contain *unique events*. Therefore, detecting multiple unique events or short unique sequences increases the confidence that you were not deceived. However, only the detection of unexpected behavior can be guaranteed.

## C. Resumption Extension

Any specification-based monitor may be extended with a resumption extension. Even a monitor that has a complete transition function may be improved by resumption, if it has unrecoverable states, like $s_\perp$ in Figure 3. Sub-scripts are used to distinguish between the original monitor ($\mathcal{L}$), the extension ($\mathcal{R}$), the extended monitor ($\mathcal{E}$) and their components respectively. $\mathrm{M}_{\mathcal{E}}$ is created by combining the sets and functions of $\mathrm{M}_{\mathcal{L}}$ with $\mathrm{M}_{\mathcal{R}}$, where $\mathrm{M}_{\mathcal{L}}$ is preferred. However, $\delta_{\mathcal{R}}$ may override $\delta_{\mathcal{L}}$ for selected verdicts, e.g., $\perp$.

*Example 1 (Resumption Extension):* Figure 8a depicts a possible extensions of the SM in Figure 2. Instead of entering a final rejecting state for unexpected events, the extended monitor ignores the event and stays in the currently active state.

The resulting $\mathrm{M}_{\mathcal{E}}$ has a complete transition function and is able to continue monitoring after reporting deviations. Thereby, the original monitor is extended by resumption.

While a resumption extension can be created in an arbitrary way, we suggest to use a resumption algorithm ($\mathcal{R}$) to create the extension. The algorithm's core function (14) takes an event and a set of (possible) active states as input. It returns the set of states that are candidates for resumption. The $\mathcal{R}$-based resumption extension can be easily exchanged to adjust the monitor to the current application scenario. Let $\mathbb{S}_C = \mathbb{S}_{\mathcal{L}} \cup \{s_{\mathcal{R}}\}$ and $\mathcal{P}(\mathbb{S}_C)$ be the set of all its subsets.

$$\mathcal{R} : \mathcal{P}(\mathbb{S}_C) \times \mathbb{E} \to \mathcal{P}(\mathbb{S}_C) \tag{14}$$

Using $\mathcal{R}$, the additional states and transitions needed for the extension of the original monitor can be derived. For finite sets $\mathbb{S}_{\mathcal{L}}$ and $\mathbb{E}$, a preparation step creates the states $\mathcal{P}(\mathbb{S}_C) \setminus \mathbb{S}_C$. The transitions are derived by evaluating $\mathcal{R}$ to find the target state. If $\mathcal{R}(q, e)$ returns an empty set or solely states that cannot reach any state in $\mathbb{S}_C$, it reached a finally non resumable state. The existence of such states depends on $\mathcal{R}$ and the specification. All states not reachable from a state in $\mathbb{S}_C$ can be pruned.

An alternative is using $\mathcal{R}$ at runtime as transition function during resumption. If $\mathcal{R}$ returns solely a single state in $\mathbb{S}_{\mathcal{L}}$, $\mathrm{M}_{\mathcal{L}}$ can resume verification in that state. Otherwise, the candidates are tracked in parallel while removing non-conforming ones. If $s_{\mathcal{R}}$ is a candidate, $\mathcal{R}$ is used to transition the candidate set, otherwise $\delta_{\mathcal{L}}$. $\gamma_{\mathcal{R}}$ is created using (5) with a suitable $\oplus$.

### *D. Resumption Algorithms*

This section introduces algorithms that can be used for resumption. These algorithms are often mimicked to extend specifications manually for creating robust monitors. Based on an observed event and a set of candidates for the active state, $\mathcal{R}$ will determine possible states of SUO with respect to the observed property. The presented algorithms can generally be categorized into *local* and *global* algorithms. The former are influenced by the state that was active before the deviation, while the latter analyze all states equally. Each of the algorithms can be used to construct a replacement for $\delta^+$ introduced in subsection IV-A and represents a different *error model*. They are only guaranteed to find all unexpected behaviors, if all occurring deviations match this error model. However, certain assumptions can lead to a significantly simpler $\mathrm{M}_{\mathcal{E}}$. The results of applying the algorithms on the SM from Figure 2 are shown in Figure 8.

The local algorithm *Waiting* (15) is inadvertently used when interpreting a trace with an unaltered state machine. Implementations usually ignore superfluous messages and remain in the same state, waiting for the next valid event. Therefore, the algorithm introduces no runtime overhead. It resumes verification with the next event accepted by the previously active state $q$, i.e., it stays in $q$ and skips all events not in $\mathbb{E}^q$. This creates loops at every state as shown in Figure 8a for the example. However, $\mathcal{R}_{\text{wait}}$ requires that all deviations are superfluous message that may be ignored. Otherwise, the guarantee to find all unexpected behaviors will no longer hold. Therefore, it is expected to perform badly for other deviations and may stall in many scenarios.

$$\mathcal{R}_{\text{wait}}(\mathbb{S}_{in}, e) = \mathbb{S}_{in} \tag{15}$$

An obvious danger is, SUO may never emit an event that is accepted by the active state. Therefore, the next algorithms also considers states around the active state. The used distance measure $\|s_s, s_t\|$ is the number of transitions $\in \delta_{\mathcal{L}}$ in the shortest path between a source state $q_s$ and a target state $q_t$. The extension $\|\mathbb{S}_s, \mathbb{S}_t\|$ is the transition count of the shortest path between any state in $\mathbb{S}_s$ and any state in $\mathbb{S}_t$. The algorithm *Nearest* (16) resumes verification with the next event that is accepted by any state reachable from the active state. Figure 8b shows the extension of the example. If multiple transitions match, it chooses the transition reachable with the fewest steps from the previously active state. A static calculation of $\mathcal{R}_{\text{near}}$ requires additional states only if tie-breakers are needed. $\mathcal{R}_{\text{near}}$ assumes that the deviations will be caused by skipped messages. It will resume on the next matched event, unless the two closest valid states require the same number of steps. As it only looks forward, superfluous or altered messages may cause it to errantly skip ahead.

$$\mathcal{R}_{\text{near}}(\mathbb{S}_{in}, e) = \underset{q_t \in \delta_{\mathcal{L}}(\mathbb{S}_C, e)}{\operatorname{argmin}} \min_{q_s \in \mathbb{S}_{in}} \|q_s, q_t\| \tag{16}$$

The algorithm *Nearest-or-Waiting* (17) is a combination of *Nearest* and *Waiting*. It measures the length of the shortest path from the active state to a state returned by *Nearest* and the shortest path of any candidate state with an accepting transition to the active state. If the latter path is shorter, *Waiting* is used. The extension of the example using $\mathcal{R}_{\text{n-o-w}}$ is depicted by Figure 8c. The idea is to ignore superfluous messages and identify them by looking as far back as was required to look forward to find a match.

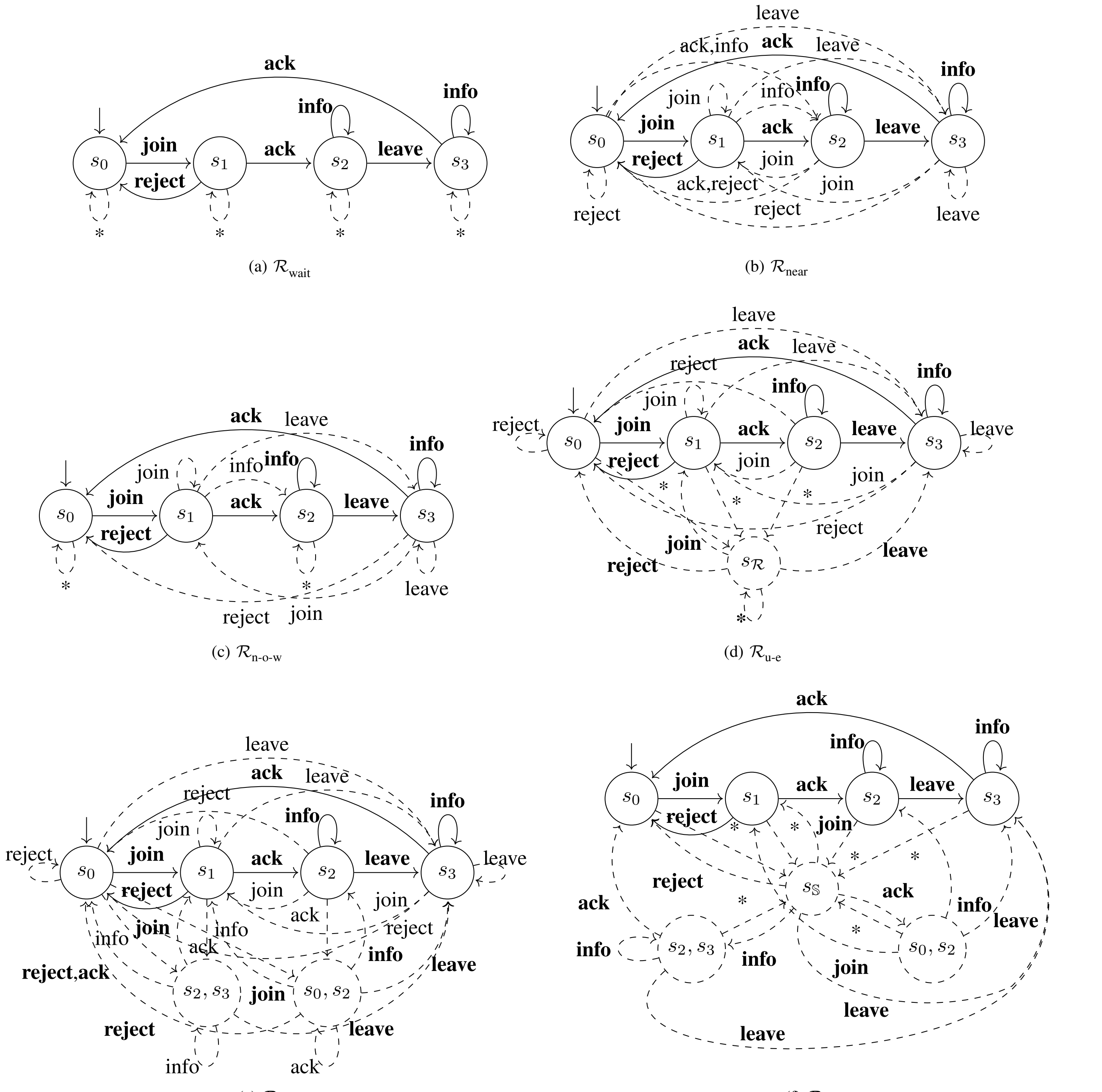
$$\mathcal{R}_{\text{n-o-w}}(\mathbb{S}_{in}, e) = \begin{cases} \mathcal{R}_{\text{wait}}, & \text{if } \|\mathbb{S}_C^e, \mathbb{S}_{in}\| < \|\mathbb{S}_{in}, \mathcal{R}_{\text{near}}\| \\ \mathcal{R}_{\text{near}}, & \text{otherwise} \end{cases} \tag{17}$$

Global algorithms consider the whole specification to identify the current communication state. Therefore, they analyze all states equally to keep all options open for resumption.

*Unique-Event* (18) resumes verification if the event is unique, i.e., the event is used on transitions to a single state only. $\mathcal{R}_{\text{u-e}}$ is the only examined $\mathcal{R}$ that ignores all input states. As there is only one target state of a unique event in the state machine, the algorithm considers this a *synchronization point*. For other events, a resumption state ($s_{\mathcal{R}}$) is entered. This extension is illustrated for the example in Figure 8d. Therefore, a static calculation only needs a single additional state.

$$\mathcal{R}_{\text{u-e}}(\mathbb{S}_{in}, e) = \begin{cases} \delta_{\mathcal{L}}(\mathbb{S}_C, e), & \text{if } |\delta_{\mathcal{L}}(\mathbb{S}_C, e)| = 1 \\ \{s_{\mathcal{R}}\}, & \text{otherwise} \end{cases} \tag{18}$$

*Unique-Sequence* (19) extends the previous algorithm to unique sequences of events, as unique events may not be available or regularly observable in every specification. $\mathcal{R}_{\text{u-s}}$ follows all valid paths simultaneously and resumes verification if there remains exactly one target state for an observed sequence. Similar to homing sequences used in model-based testing, $\mathcal{R}_{\text{u-s}}$ aims to reduce the current state uncertainty with each step. It evaluates which of the input states accept the event. If the observed event is part of a separating sequence, the non-matching states are removed from the set. If a merging sequence was found, the following $\delta_{\mathcal{L}}$-step returns the same state for two input states and the number of candidates is further reduced. If there are homing sequences for $\mathcal{L}$ and the



(a) $\mathcal{R}_{\text{wait}}$ (b) $\mathcal{R}_{\text{near}}$

(c) $\mathcal{R}_{\text{n-o-w}}$ (d) $\mathcal{R}_{\text{u-e}}$

(e) $\mathcal{R}_{\text{u-s}}$ (f) $\mathcal{R}_{\text{e-b}}$

Figure 8. SMs with states and transitions added (dashed) by different $\mathcal{R}$. **Bold** labels indicate an accepting and regular labels a rejecting verdict. The wild-card '*' matches all events that have no other transition in the state.

SUO emits one, $\mathcal{R}_{\text{u-s}}$ will detect it. Any unexpected behavior causes $\delta_{\mathcal{L}}(\mathbb{S}_{in}, e)$ to be empty and therefore resets the set of possible candidates to any state accepting the event, i.e., the resumption is reset. New intermediate states are created to capture the current state uncertainty during resumption. For example, '$s_2, s_3$' in Figure 8e means that $\mathrm{M}_{\mathcal{E}}$ considers the SUO in state $s_2$ or $s_3$. Therefore, the size of the static calculation is limited by $|\mathrm{SM}^{\mathcal{P}}|$.

$$\mathcal{R}_{\text{u-s}}(\mathbb{S}_{in}, e) = \begin{cases} \delta_{\mathcal{L}}(\mathbb{S}_{in}, e), & \text{if } \delta_{\mathcal{L}}(\mathbb{S}_{in}, e) \neq \emptyset \\ \delta_{\mathcal{L}}(\mathbb{S}_{C}, e), & \text{else if } \delta_{\mathcal{L}}(\mathbb{S}_{C}, e) \neq \emptyset \\ \{s_{\mathcal{R}}\}, & \text{otherwise} \end{cases} \quad (19)$$

The formal analysis in this extended version of [1] has shown that $\mathcal{R}_{\text{u-s}}$ is not the most general case. Like the other algorithms, it can be used to extract trace segments containing deviations matching its error model. However, resumption with $\mathcal{R}_{\text{u-s}}$ uses the last event of the previous unexpected behavior. Hence, verification may be deceived as this reflects a deviation to a state accepting the event instead of any state. Therefore, we introduce the algorithm *Expected-Behavior* (20) that takes into account all expected behavior. It is the resumption algorithm version of the candidate selection $\delta^{+}$ given in (8). As shown in Figure 8f, $s_{\mathbb{S}}$ is always entered in case of a violation. This reflects the assumption that the SUO could be in any state after a deviation. Theorem 7 indicates that multiple unique sequences of expected behavior can further improve a monitors confidence when identifying deviations. $\mathcal{R}_{\text{2-e-b}}$ uses 2 of such sequences.

$$\mathcal{R}_{\text{e-b}}(\mathbb{S}_{in}, e) = \begin{cases} \delta_{\mathcal{L}}(\mathbb{S}_{in}, e), & \text{if } \delta_{\mathcal{L}}(\mathbb{S}_{in}, e) \neq \emptyset \\ \mathbb{S}_{C}, & \text{otherwise} \end{cases} \quad (20)$$

A variety of resumption algorithms have been introduced. And this enumeration is not conclusive. More could be created to match specific requirements. The next section presents an evaluation framework that identifies strength and weaknesses in the algorithms' performance. It can be used to judge the algorithm for a given application scenario and identify the best.

## V. Evaluation

This section presents an evaluation of the introduced method for automatic resumption of runtime verification monitors. We have already given proof that all unexpected behaviors can always be found and used to encircle the detectable deviations in Theorems 2 and 5. They are also confirmed by the collected raw data. The evaluation examines how well the presented resumption algorithms perform for different SMs and kinds of deviations. If the monitor's uncertainty of the SUO's state is reduced to a single state without missing a deviation, a detected unexpected behavior equals a deviation. Therefore, if $\mathrm{M}_{\mathcal{E}}$ can identify deviations well, it performs a good resumption.

### A. Evaluation Framework

An overview of the evaluation setup is depicted in Figure 9. The framework is employed to compare the presented resumption algorithms. A specific *application scenario* usually provides the specification and, thereby, a *reference model*. However, to make statements about the general performance of the algorithms, a generator creates the models. A single state is used as initial seed. In each iteration step, one state is randomly selected and replaced by a randomly chosen number of states. The input and output transitions of the replaced state are assigned to random states in the subset. The subset of states is then connected with transitions, guaranteeing that each state is reachable from an incoming transition and can reach an outgoing transition of the original state. Otherwise, the states are connected randomly. The used set of events changes with each iteration step, i.e., some events are removed or some new events are added. This is repeated to create SMs of various sizes. The resulting SMs use global events across the whole machine and local groups. To classify the SMs, different metrics of their structure are collected, e.g., number of states and uniqueness. *Uniqueness* is the likelihood of an occurring event being unique. It is approximated by the fraction of all transitions in the SM that have a unique event.

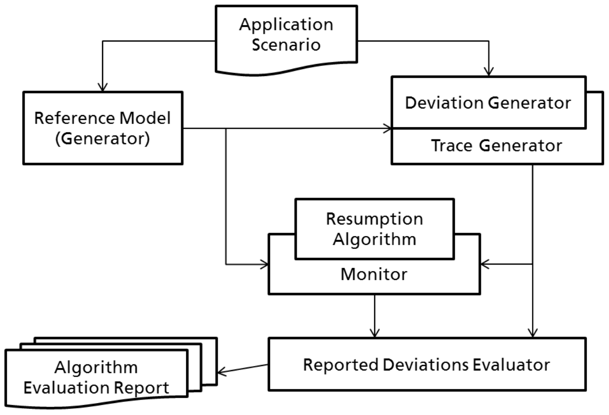

Application Scenario
Reference Model (Generator)
Deviation Generator
Trace Generator
Resumption Algorithm
Monitor
Reported Deviations Evaluator
Algorithm Evaluation Report

Figure 9. Overview of the evaluation framework for resumption algorithms.

For each reference model, multiple traces are generated by randomly selecting paths from $\mathrm{SM}'$. The *deviation generator* creates $\mathrm{SM}'$ by adding new states and transitions to SM for the deviations shown in Figure 10. The added transitions use undefined events ($\chi \notin \mathbb{E}^{q_s}$) of the source state $q_s$. The different deviations are characterized by their choice of transition targets $q_t$: superfluous ($q_t = q_s$), altered ($\exists e : \delta_{\mathcal{L}}(q_s, e) \mapsto q_t$), skipped ($\exists e : \delta_{\mathcal{L}}(q_s, e) \mapsto q_i \wedge \delta_{\mathcal{L}}(\chi, q_i) \mapsto q_t$) and random events ($q_t \in \mathbb{S}_{\mathcal{L}}$). The generated transitions for deviations are equivalent to faults in a real implementation. If a scenario expects more complex deviations, they can be simulated by combining these deviations. However, to evaluate the influence of each deviation kind, we apply only one kind of deviation per trace. For later analyses, the injected deviations are marked in the meta-data of the trace, which is invisible to the monitor.

The traces are eventually checked using SM extended with the individual $\mathcal{R}$. For the evaluation, our Eclipse-based tool DANA was used and extended. It is capable of using reference models as monitors [7]. Using hooks in its model execution runtime, resumption is injected if needed. Thereby, all introduced algorithms can easily be exchanged. In addition, we include $\mathcal{R}_{\text{l-c}}$, an offline *least changes* (see Section III) algorithm, for comparison.

The goal of the evaluation framework is to measure how well a monitor is at finding multiple deviations in a given application scenario. Therefore, the *reported deviations evaluator* rates each algorithm's performance by comparing the

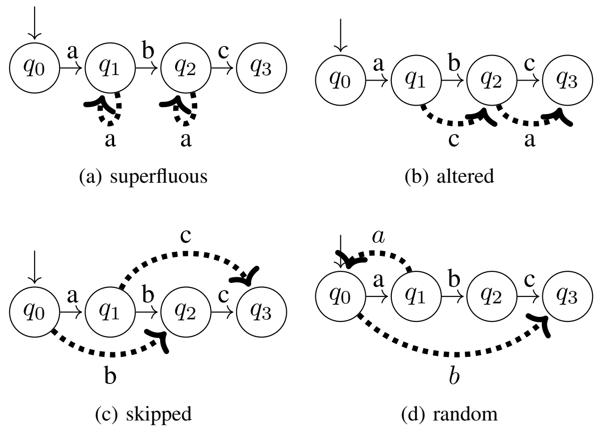


Figure 10. Examples illustrating the different deviation modules used in evaluation. The deviations are shown by dotted arrows.

detected and the injected deviations. It calculates for each extended monitor the well-established metrics from information retrieval: *precision* and *recall* [28][29]. Precision (21) is the fraction of reported deviations ($rd$) that were true ($td$), i.e., injected by the deviation generator. Recall (22) is the fraction of injected deviations that were reported. Both values are combined to their harmonic mean, also known as $F_1$ score (23).

$$p = |td \cap rd|/|rd| \quad (21)$$

$$r = |td \cap rd|/|td| \quad (22)$$

$$F_1 = 2 \cdot \frac{p \cdot r}{p + r} \quad (23)$$

A monitor that reports only and all true deviations has a perfect precision $p = 1$ and recall $r = 1$. Up to the first deviation, all extended monitors exhibit this precision, as they work like regular monitors in this case. Regular monitors only maintain this precision by ignoring everything that follows. Extended monitors may lose precision as they attempt to find further deviations. Therefore, recall estimates how likely all true deviations are reported. A regular monitor reports only the first deviation; thus, its recall is $|td|^{-1}$.

## B. *Comparison of Resumption Algorithms*

The subscription service example (cf. Figure 2) evaluates to the $F_1$ scores: $\mathcal{R}_{\text{wait}} \mapsto 0.56$, $\mathcal{R}_{\text{near}} \mapsto 0.71$, $\mathcal{R}_{\text{n-o-w}} \mapsto 0.82$, $\mathcal{R}_{\text{u-e}} \mapsto 0.82$, $\mathcal{R}_{\text{u-s}} \mapsto 0.81$, $\mathcal{R}_{\text{e-b}} \mapsto 0.99$, $\mathcal{R}_{\text{2-e-b}} \mapsto 0.99$,$\mathcal{R}_{\text{l-c}} = 0.93$. For the general evaluation, traces with a total of 80 million deviations in 220 different SMs with up to 360 states have been generated and were analyzed by monitors extended with the algorithms. Each trace included 20 injected deviations on average, so the recall for a monitor reporting only the first deviation is 0.05 and its $F_1$ score 0.095. Figure 11 shows the precision and recall for each $\mathcal{R}$ per kind of deviation. While $\mathcal{R}_{\text{wait}}$ has the worst precision for most deviations, it shows very high recall scores overall and a perfect result for superfluous deviations. Besides that, each algorithm performs very similar for altered and superfluous deviations. When comparing $\mathcal{R}_{\text{near}}$ and $\mathcal{R}_{\text{n-o-w}}$, the former has slightly less precision; however, it provides a better recall. $\mathcal{R}_{\text{u-e}}$ has a low recall independent of the kind of deviation but also a good precision for skipped deviations. Overall, $\mathcal{R}_{\text{u-s}}$ has a high recall. The offline algorithm $\mathcal{R}_{\text{l-c}}$ has a perfect result for superfluous and altered deviations. For skipped deviations, it also provides the missing event very reliably. While hardly visible at the scale of the figure, $\mathcal{R}_{\text{e-b}}$'s precision of 0.9878 was improved to 0.9995 by a second unique sequence in $\mathcal{R}_{\text{2-e-b}}$.

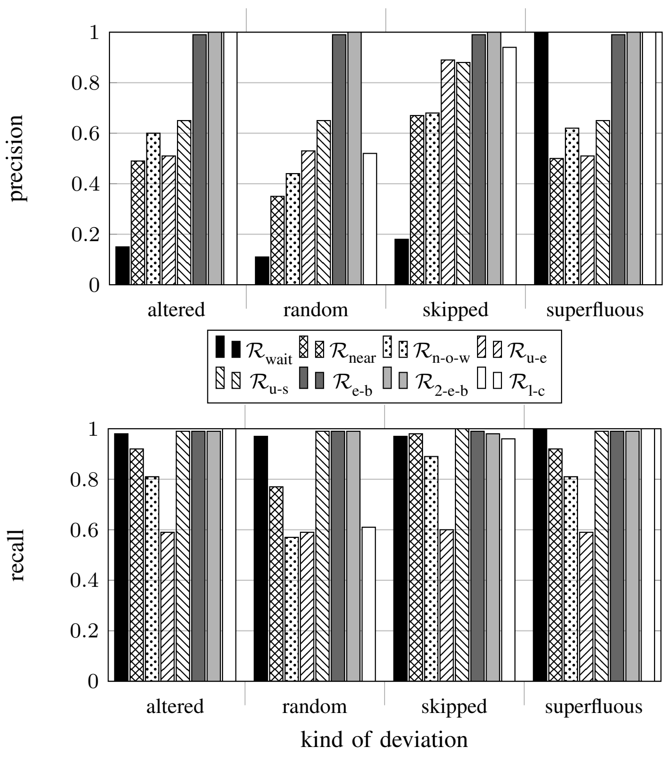


Figure 11. Precision and recall of $\mathcal{R}$ compared for different kinds of deviations.

Figure 12 compares the $F_1$ scores of the algorithms for different levels of uniqueness and numbers of states of the generated SMs. For clarity, SMs are grouped into buckets based on the metrics and the scores are averaged for each bucket. This allows a quick comparison of the algorithms, but hides the distribution of the scores across the evaluated SMs. These details are visible in the scatter plots in the Appendix, Figure 16. For example, $\mathcal{R}_{\text{2-e-b}}$ performed nearly perfect for all tested scenarios, whereas the results of $\mathcal{R}_{\text{near}}$ are much more distributed. The low overall score of $\mathcal{R}_{\text{wait}}$ is clearly visible for both metrics. For SMs with low uniqueness, $\mathcal{R}_{\text{u-s}}$ outperforms many other algorithms. However, its $F_1$ score slightly drops with increased uniqueness. The other algorithms benefit from an increase of uniqueness, especially $\mathcal{R}_{\text{u-e}}$. For very high uniqueness, $\mathcal{R}_{\text{u-s}}$ and $\mathcal{R}_{\text{u-e}}$ are identical. Nevertheless, both $\mathcal{R}_{\text{n-o-w}}$ and $\mathcal{R}_{\text{near}}$ perform better in this case. An increase of the state count leads to a declined performance for $\mathcal{R}_{\text{u-e}}$, $\mathcal{R}_{\text{n-o-w}}$ and $\mathcal{R}_{\text{near}}$. $\mathcal{R}_{\text{u-e}}$ even drops below $\mathcal{R}_{\text{wait}}$. $\mathcal{R}_{\text{e-b}}$ and $\mathcal{R}_{\text{2-e-b}}$ are hardly affected by state count and uniqueness and provide a near perfect overall performance. The slight advantage of $\mathcal{R}_{\text{2-e-b}}$ can be seen by the small decline of $\mathcal{R}_{\text{e-b}}$ for low uniqueness and high state counts. $\mathcal{R}_{\text{l-c}}$ cannot match this performance, but still excels the remaining algorithms. For higher state counts, its score is comparable to $\mathcal{R}_{\text{u-s}}$.

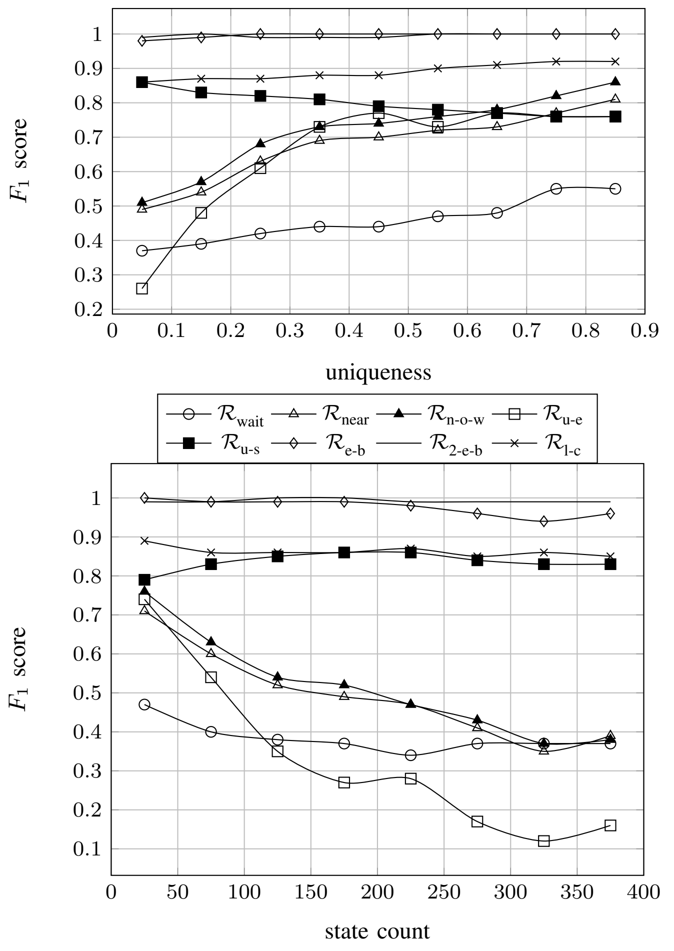


Figure 12. $F_1$ scores of $\mathcal{R}$ compared for metrics uniqueness and state count.

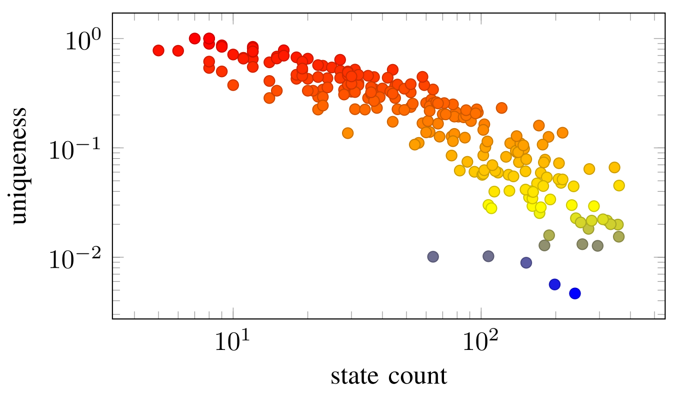


Figure 13. Scatter plot comparing uniqueness and state count of the state machines used for evaluation.

## C. Discussion

In this evaluation, $\mathcal{R}_{\text{1-c}}$ is outperformed by $\mathcal{R}_{\text{e-b}}$ and $\mathcal{R}_{\text{2-e-b}}$. They perform almost perfectly for all tested scenarios. So does $\mathcal{R}_{\text{1-c}}$ for deviations matching its edits. However, it is challenged by random deviations. For example, a random deviation can transition out of a dead end or a completely different part of SM. $\mathcal{R}_{\text{1-c}}$ can only match this, if it steps back in its search space and introduces further edits. In contrast, $\mathcal{R}_{\text{e-b}}$ and $\mathcal{R}_{\text{2-e-b}}$ can match such behaviors. The slight improvement with $\mathcal{R}_{\text{2-e-b}}$ also shows that a higher precision for detecting deviations can be reached by requiring more than a single unique sequence to resume verification. While we could provide an upper bound for the worst case space and time requirements of $\mathcal{R}_{\text{e-b}}$, which is equal to segmenting the trace by unexpected behaviors, sometimes more efficient algorithms are desired.

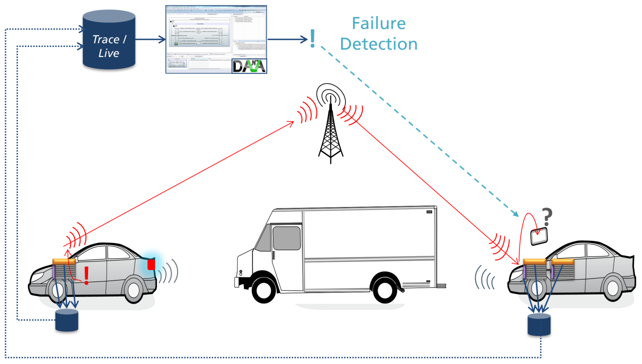
The perfect precision and recall of $\mathcal{R}_{\text{wait}}$ for superfluous deviations were as expected, since this deviation matches exactly the resumption behavior of the algorithm. This shows that knowing the kind of deviation expected in a scenario can help formulate specialized, highly efficient algorithms. However, $\mathcal{R}_{\text{wait}}$ performs worst for all other kinds of deviations, as the SUO transitioned already internally to a different state and would have to return to its original state. It benefits from unique events, because they prevent taking wrong transitions in the meantime.

The metric uniqueness can be used as indication for the class of algorithm that is needed for a scenario. For low values, the algorithm needs to combine multiple events in order to reliably synchronize model and SUO. Therefore, in this case, algorithms should be preferred that take multiple events into account, e.g., $\mathcal{R}_{\text{u-s}}$. However, $\mathcal{R}_{\text{u-s}}$ slightly drops its precision with increasing uniqueness, as the chance increases to overeagerly synchronize with an erroneous unique event. For example, if all events are unique, any observed deviation is a unique event and the algorithm will resume with the associated state. As the next valid event is unique again, the monitor will jump back. However, in this case it registered two deviations when there actually was only one. The same holds for $\mathcal{R}_{\text{u-e}}$. Therefore, especially with a high uniqueness, it may be desirable to limit the number of options for which an algorithm may resume and use a local resumption algorithm instead. The choice between $\mathcal{R}_{\text{near}}$ and $\mathcal{R}_{\text{n-o-w}}$ depends on the desired precision and recall. According to the $F_1$ score, $\mathcal{R}_{\text{n-o-w}}$ is slightly favorable. However, as these algorithms may maneuver themselves into dead-ends, they are less suited for higher state counts. A bias towards lower uniqueness for higher state counts in the sample set severs the impact on $\mathcal{R}_{\text{u-e}}$. This bias is indicated by the diagonal arrangement in the comparison of uniqueness and state count for the evaluated SMs depicted in Figure 13. Nevertheless, in all cases, the $F_1$ scores of the extended monitors always show better results than for a regular monitor.

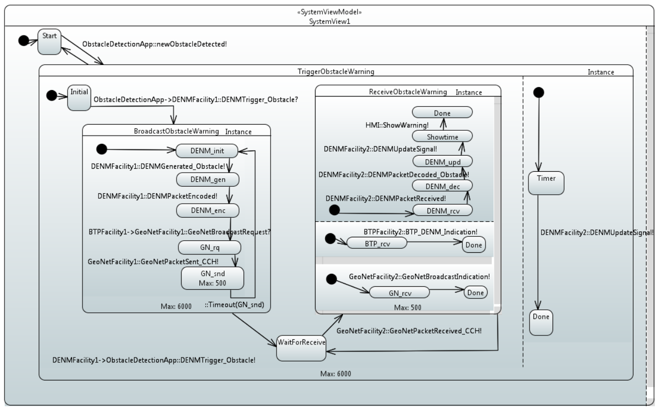
The results for the subscription service example (uniqueness 0.43, 4 states) and the respective results from Figure 12 match well. While the evaluation framework can be used to identify the best suited algorithm, this example shows that the metrics state count and uniqueness can be used as indicators for such a selection.

## VI. Real World Application Scenarios

In this section, we provide examples where resumption has been applied successfully to real world use cases. DANA is the platform for description and analysis of networked applications that the presented resumption concepts were implemented in. Previous work has already shown how the platform can be used to analyze various in-vehicle infotainment functions at runtime, e.g., an *auxiliary input* service [7], and a *parking assistance* service [30]. In both cases, preliminary versions of the resumption algorithms were employed.

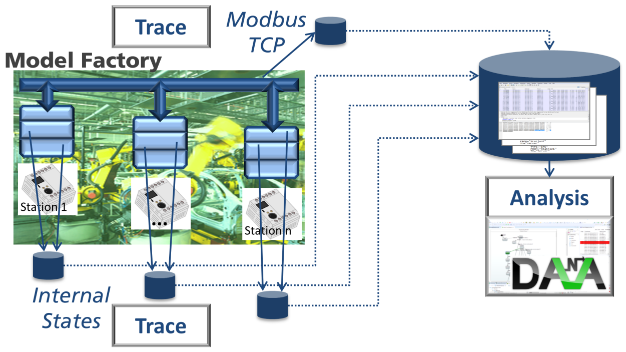


(a) Illustration of the use case.

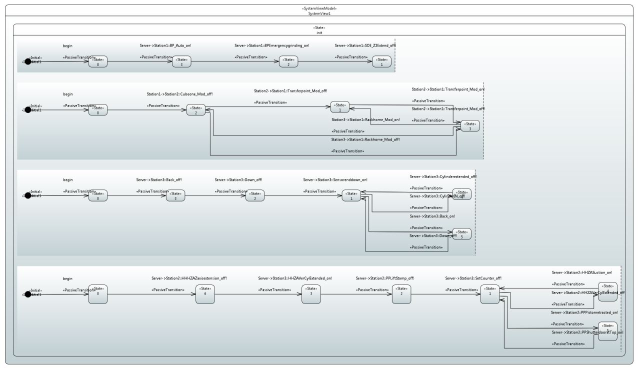

«SystemViewModel» SystemView1
Start
ObstacleDetectionApp::newObstacleDetected!
TriggerObstacleWarning
Instance
Initial
ObstacleDetectionApp->DENMFacility1::DENMTrigger_Obstacle?
BroadcastObstacleWarning Instance
DENM_init
DENMFacility1::DENMGenerated_Obstacle!
DENM_gen
DENMFacility1::DENMPacketEncoded!
DENM_enc
BTPFacility1->GeoNetFacility1::GeoNetBroadcastRequest?
GN_rq
GeoNetFacility1::GeoNetPacketSent_CCH!
GN_snd
Max: 500
Max: 6000
::Timeout(GN_snd)
ReceiveObstacleWarning Instance
Done
HMI::ShowWarning!
Showtime
DENMFacility2::DENMUpdateSignal!
DENM_upd
DENMFacility2::DENMPacketDecoded_Obstacle!
DENM_dec
DENMFacility2::DENMPacketReceived!
DENM_rcv
BTPFacility2::BTP_DENM_Indication!
BTP_rcv
Done
GeoNetFacility2::GeoNetBroadcastIndication!
GN_rcv
Done
Max: 500
Timer
DENMFacility2::DENMUpdateSignal!
Done
WaitForReceive
GeoNetFacility2::GeoNetPacketReceived_CCH!
DENMFacility1->ObstacleDetectionApp::DENMTrigger_Obstacle!
Max: 6000

(b) Behavioral model for the use case.

Figure 14. Use case of a hazard warning application.



(a) Illustration of the use case.



(b) Behavioral model for the use case.

Figure 15. Use case of a small industrial plant.

In a more recent use case, we successfully employ resumption to support the development of a *hazard warning application*. A sudden obstacle in traffic can be dangerous; especially, if drivers realize the obstacle too late. A hazard warning can help to inform drivers in time. For example, in Figure 14a the driver in the car on the left notices an obstacle and brakes hard. As the view of the right car's driver is occluded by the van in the middle, she would only be able to notice the reaction of the van in front. With a hazard warning message from the front car, she could start braking immediately. However, such an application involves multiple cars, thus, multiple systems have to be considered. DANA is also capable to address such distributed networked systems, e.g., connected cars. We can capture the behavior to be verified of all involved cars in a model and use this for verification. Hooks in the used communication stack for car-to-car communication are employed to monitor the different communication layers. Thereby, deviations in the hazard warning implementations can be identified. For instance, the reason why a hazard warning was not displayed in the receiving car can easily be located by monitoring the progress of the animated state machine. Resumption helps to analyze the underlying problem: By providing a model containing the expected interactions, all violations of a single run can be identified.

Besides the automotive domain, resumption has been successfully applied to other application scenarios. Figure 15a shows a small industrial plant composed of three stations. The plant assembles cubes from two halves. The first station collects parts from two magazines and checks their orientation and material. The second station joins the two halves in a hydraulic press. The third station stores the assembled cubes. For each station, the changes of internal sensors and actuators controlled by the respective station are reported. This enables monitoring the plant's operation without having to alter the original control program. The communication between the stations is recorded by tapping into the switch of the Ethernet-based Modbus TCP connection. Actually, the model shown in Figure 15b contains much more details as it was automatically learned from observed behavior, i.e., each of the states contains sub-states, which are hidden in this example for clarity. Nevertheless, the sub-states are still used for verification, while this diagram provides a comprehensive overview of the plant's overall operation. Resumption helps to overcome imperfections, which such a learned model may have. While the monitor will report unexpected behavior in the case of an imperfection, resumption can often realign model and system so that verification can continue. A developer analyzing the (falsely) reported unexpected behaviors can use this feedback to improve the behavior description.

## VII. Conclusion

To conclude the paper, the contributions are summarized and the findings are discussed before an outlook to future work is provided.

### A. Contributions

This work examined the identification of all differences between a trace of a system under observation (SUO) and its specification at runtime through resumable monitoring. We have shown under which conditions deviations of the SUO can be detected and when they will be missed. As the main problem of detecting deviations is the current state uncertainty, the detection of unexpected behavior was examined. Unexpected behavior is independent of the SUO's actual state. By definition, the verification only needs to consider possible states of the SUO. We could show that all occurrences of unexpected behavior can be found within space $O(|\mathrm{SM}|)$ and time $O(|\mathbb{S}|)$ per step at runtime. Such unexpected behavior is always an indication for a deviation, but there may still be deviations that cannot be detected with the available information. Using these results, we have introduced a method for extending runtime monitors with resumption. Such an extension allows a specification-based monitor to find subsequent deviations. Thereby, an existing reference model of the system can be used directly without creating a secondary specification for test purposes only. Each of the introduced resumption algorithms has its strength and weaknesses. The presented framework and metrics help to find the best suited algorithm for an application scenario. Nevertheless, *expected-behavior* is the most general case of a resumption algorithm, as it has the least assumptions on possible deviations. Identifying all unexpected behaviors guarantees that all subsequences containing detectable deviations are reported.

By the result of the evaluation, $\mathcal{R}_{\text{e-b}}$ that tests for any expected behavior is the most stable and reliable of the compared algorithms. Yet, the evaluation result is not surprising, considering it is the resumption algorithm equivalent to the general candidate function $\delta^+$ used in subsection IV-A. Therefore, using runtime verification with $\mathcal{R}_{\text{e-b}}$ resumption is equal to segmenting a trace by unexpected behaviors for any kind of deviation. Further, this fulfills our main research goal of reporting all detectable deviations using a single monitor instance.

### B. Discussion

The biggest misery for resumption shown in this paper is that the current state uncertainty can only be provably reduced by an obviously conforming merging sequence and there is no guarantee that such a sequence exists. While the evaluation has demonstrated that for many cases it is still possible to identify deviations with a high likelihood using resumption, there may be cases where it is impossible to identify deviations. The problem is not limited to resumption. How is this handled by other approaches? Some consider certain events or sequences to be trusted, e.g., model-based testing can rely on the input it provides to the SUO. A fixed initial state is another example. If such events are integrated into the specification, they can be used as a kind of checkpoint to reliably resume verification. Other approaches may only attempt to detect unexpected behavior. There are runtime verification approaches that expect the specification to be split into multiple properties. They rely on detecting trigger sequences before they verify a constraint. However, their verdict always judges trigger and constraint. Further, these triggers can get quite complex and, for example, check for necessary or sufficient conditions of the property. With many properties to test, these checks may quickly become redundant. Therefore, it may be more efficient to use a single state machine, as we could show that it can detect all unexpected behaviors. Different verdicts can be associated with missing transitions and, thereby, used to retain the categorization of unexpected behaviors provided by a set of properties.

Resumption strives to resume verification after a deviation was observed. In general, the quality of the resumption depends on carefully selecting candidates for the actual state of the SUO. This entails a strong relation to the deviations that are possible or expected. Different assumptions on the system, e.g., the kind of deviations, will lead to different selections of optimal candidates. The selection of candidates is made by a *resumption algorithm*. This replaces $\delta^+$ from (8) with a different function. $\delta^+$ determines the current state uncertainty for the next step. Therefore, all other findings in this paper remain untouched, even if a different algorithm is chosen. However, any deviation that does not match the candidates provided by $\delta^+$, breaks the guarantee of detecting all unexpected behaviors. Nevertheless, this allows to use specialized algorithms for specific application scenarios. Several different resumption algorithms have been evaluated and compared with regard to how well they detect deviations. $\mathcal{R}_{\text{wait}}$ is often (unintentionally) used, as it just ignores deviating events and waits in the same state. For the general case, this is not always correct and usually very unreliable. However, if deviations are only superfluous - the SUO stays in the same state when deviating - $\mathcal{R}_{\text{wait}}$ provides perfect identification of deviations. The proof is simple, as the current state uncertainty always contains exactly one state. Thereby, we can apply Theorem 1.

### C. Future Work

Currently, the resumption algorithms are designed for plain state machines. This is still useful, as many advanced design concepts for state machines, like hierarchical state machines and orthogonal regions can be directly mapped to plain state machines. The event model in the layered reference model allows the state machine to be oblivious to parameter values, which would otherwise require extended state machines. If the event model can store parameters for comparison, this implies there is a state of the event model in the parameter space in addition to the state of the state machine. However, the event model is currently not updated by resumption. This is the equivalent of using $\mathcal{R}_{\text{wait}}$ for states. Therefore, future work will extend resumption to include the parameter space, i.e., the event model.

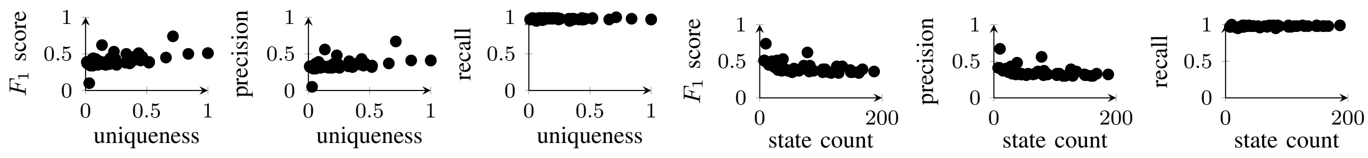
Moreover, we currently expect a total order on the observed events. In a distributed system, events can be collected in independent traces at many different sources and a global time is not always available. Therefore, obtaining a total order for all observations is not always feasible. The reference model needs to be extended to better support modeling such behavior. Possibly, a special kind of resumption can be utilized to synchronize the different traces.

## Acknowledgment

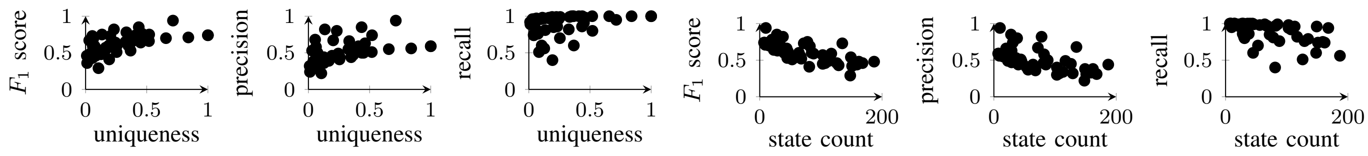
The work leading to these results has been partially funded by the Bavarian Ministry of Economic Affairs, Infrastructure, Transport and Technology.

## Appendix
## Detailed Results from Evaluation

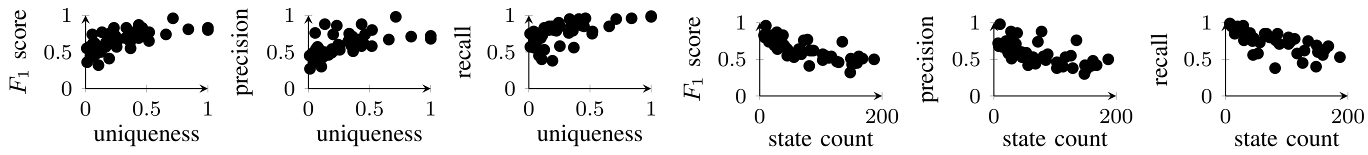
The scatter-plots in Figure 16 show the average results of the presented resumption algorithms for each machine.



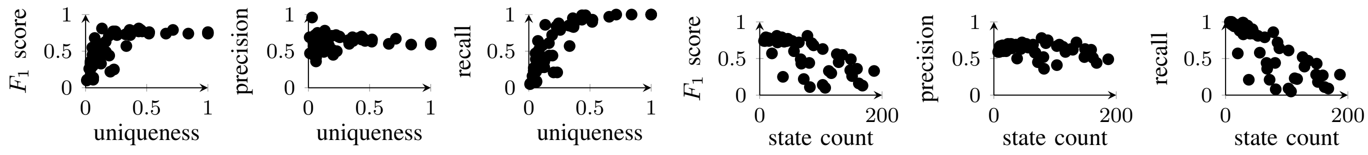
(a) $\mathcal{R}_{\text{wait}}$



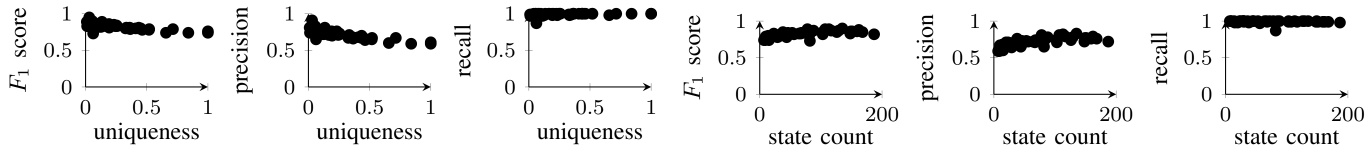
(b) $\mathcal{R}_{\text{near}}$



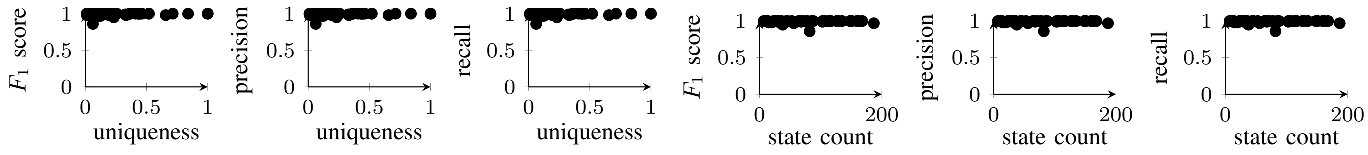
(c) $\mathcal{R}_{\text{n-o-w}}$



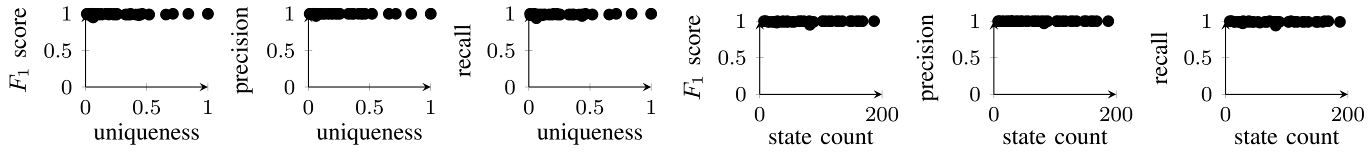
(d) $\mathcal{R}_{\text{u-e}}$



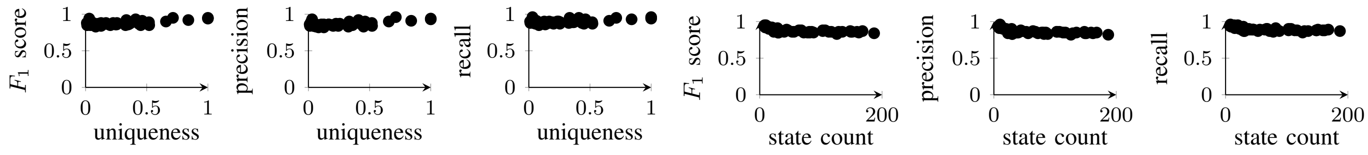
(e) $\mathcal{R}_{\text{u-s}}$



(f) $\mathcal{R}_{\text{e-b}}$



(g) $\mathcal{R}_{\text{2-e-b}}$



(h) $\mathcal{R}_{\text{l-c}}$

Figure 16. Scatter plots of the algorithms' $F_1$ score, precision and recall on the y axis. The x axis of the three left plots shows uniqueness, the right ones' show the state count of the evaluated machine. Each dot represents the result of all evaluations for one machine with the respective algorithm.

## References

[1] C. Drabek, G. Weiss, and B. Bauer, "Method for automatic resumption of runtime verification monitors," in SOFTENG 2017, The Third International Conference on Advances and Trends in Software Engineering, Venice, Italy, Apr. 2017, pp. 31–36. [Online]. Available: http://www.thinkmind.org/index.php?view=article&articleid=softeng_2017_2_20_64084

[2] A. Hagemann, G. Krepinsky, and C. Wolf, "Interface construction, deployment and operation  a mystery solved," International Journal on Advances in Software, vol. 10, no. 1, 2017, pp. 61–78. [Online]. Available: https://www.thinkmind.org/index.php?view=article&articleid=soft_v10_n12_2017_5

[3] D. Heffernan, C. Macnamee, and P. Fogarty, "Runtime verification monitoring for automotive embedded systems using the ISO 26262 functional safety standard as a guide for the definition of the monitored properties," IET Software, vol. 8, no. 5, Oct. 2014, pp. 193–203. [Online]. Available: https://doi.org/10.1049/iet-sen.2013.0236

[4] R. Rabiser, S. Guinea, M. Vierhauser, L. Baresi, and P. Grnbacher, "A comparison framework for runtime monitoring approaches," Journal of Systems and Software, vol. 125, Mar. 2017, pp. 309–321. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216302618

[5] M. Leucker and C. Schallhart, "A brief account of runtime verification," The Journal of Logic and Algebraic Programming, vol. 78, no. 5, Mai. 2009, pp. 293–303. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1567832608000775

[6] N. Delgado, A. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," IEEE Transactions on Software Engineering, vol. 30, no. 12, Dec. 2004, pp. 859–872. [Online]. Available: https://doi.org/10.1109/TSE.2004.91

[7] C. Drabek, A. Paulic, and G. Weiss, "Reducing the verification effort for interfaces of automotive infotainment software," SAE International, Warrendale, PA, SAE Technical Paper 2015-01-0166, Apr. 2015. [Online]. Available: http://papers.sae.org/2015-01-0166/

[8] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," ACM Trans. Softw. Eng. Methodol., vol. 20, no. 4, Sep. 2011, pp. 14:1–14:64. [Online]. Available: http://doi.acm.org/10.1145/2000799.2000800

[9] Y. Falcone, K. Havelund, and G. Reger, "A tutorial on runtime verification." Engineering Dependable Software Systems, vol. 34, 2013, pp. 141–175. [Online]. Available: http://ebooks.iospress.nl/publication/33757

[10] C. Allan et al., "Adding trace matching with free variables to AspectJ," in Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 345–364. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094839

[11] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Rou, "An overview of the MOP runtime verification framework," International Journal on Software Tools for Technology Transfer, vol. 14, no. 3, Apr. 2011, pp. 249–289. [Online]. Available: http://link.springer.com/article/10.1007/s10009-011-0198-6

[12] W. van der Aalst, A. Adriansyah, and B. van Dongen, "Replaying history on process models for conformance checking and performance analysis," Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, vol. 2, no. 2, Mar. 2012, pp. 182–192. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/widm.1045/abstract

[13] J. E. Cook, C. He, and C. Ma, "Measuring behavioral correspondence to a timed concurrent model," in IEEE International Conference on Software Maintenance, 2001. Proceedings. Florence, Italy: IEEE, 2001, pp. 332–341. [Online]. Available: https://doi.org/10.1109/ICSM.2001.972746

[14] G. Reger, "Suggesting edits to explain failing traces," in Runtime Verification. Springer, 2015, pp. 287–293. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-23820-3_20

[15] C. Allauzen and M. Mohri, "3-way composition of weighted finite-state transducers," in International Conference on Implementation and Application of Automata. Springer, 2008, pp. 262–273. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-70844-5_27

[16] ——, "Linear-space computation of the edit-distance between a string and a finite automaton," arXiv:0904.4686, Apr. 2009. [Online]. Available: https://arxiv.org/abs/0904.4686

[17] J. E. Cook and A. L. Wolf, "Software process validation: Quantitatively measuring the correspondence of a process to a model," ACM Trans. Softw. Eng. Methodol., vol. 8, no. 2, Apr. 1999, pp. 147–176. [Online]. Available: http://doi.acm.org/10.1145/304399.304401

[18] A. Pretschner and M. Leucker, "Model-based testing  a glossary," in Model-Based Testing of Reactive Systems, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2005, pp. 607–609. [Online]. Available: https://link.springer.com/chapter/10.1007/11498490_27

[19] T. Herpel, T. Hoiss, and J. Schroeder, "Enhanced simulation-based verification and validation of automotive electronic control units," in Electronics, Communications and Networks V, ser. Lecture Notes in Electrical Engineering, A. Hussain, Ed. Springer Singapore, 2016, no. 382, pp. 203–213. [Online]. Available: http://link.springer.com/chapter/10.1007/978-981-10-0740-8_24

[20] A. Kurtz, B. Bauer, and M. Koeberl, "Software based test automation approach using integrated signal simulation," in SOFTENG 2016, The Second International Conference on Advances and Trends in Software Engineering, Feb. 2016, pp. 117–122. [Online]. Available: http://www.thinkmind.org/index.php?view=article&articleid=softeng_2016_5_20_65040

[21] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," in Machine Learning: From Theory to Applications. Springer, Berlin, Heidelberg, 1993, pp. 51–73. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-56483-7_22

[22] S. Sandberg, "Homing and synchronizing sequences," in Model-Based Testing of Reactive Systems, M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds. Springer Berlin Heidelberg, 2005, pp. 5–33. [Online]. Available: http://link.springer.com/chapter/10.1007/11498490_2

[23] S. D. Stoller et al., "Runtime verification with state estimation," in Runtime Verification, S. Khurshid and K. Sen, Eds. Springer Berlin Heidelberg, 2012, pp. 193–207. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-29860-8_15

[24] M. Chupilko and A. Kamkin, "Runtime verification based on executable models: On-the-fly matching of timed traces," EPTCS, vol. 111,, Mar. 2013, pp. 67–81. [Online]. Available: http://arxiv.org/abs/1303.1010v1

[25] T. Berg, B. Jonsson, and H. Raffelt, "Regular inference for state machines with parameters," in Fundamental Approaches to Software Engineering, L. Baresi and R. Heckel, Eds. Springer Berlin Heidelberg, 2006, pp. 107–121. [Online]. Available: http://link.springer.com/chapter/10.1007/11693017_10

[26] A. Danese, T. Ghasempouri, and G. Pravadelli, "Automatic extraction of assertions from execution traces of behavioural models," in Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition. San Jose, CA, USA: EDA Consortium, 2015, pp. 67–72. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755753.2755769

[27] G. Reger, H. Barringer, and D. Rydeheard, "Automata-based pattern mining from imperfect traces," SIGSOFT Softw. Eng. Notes, vol. 40, no. 1, Feb. 2015, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/2693208.2693220

[28] D. M. W. Powers, "Evaluation: From precision, recall and f-measure to ROC, informedness, markedness and correlation," Journal of Machine Learning Technologies, vol. 2, no. 1, 2011, pp. 37–63. [Online]. Available: http://dspace2.flinders.edu.au/xmlui/handle/2328/27165

[29] S. Yingchareonthawornchai, D. N. Nguyen, V. T. Valapil, S. S. Kulkarni, and M. Demirbas, "Precision, recall, and sensitivity of monitoring partially synchronous distributed systems," in International Conference on Runtime Verification. Springer, Cham, Sep. 2016, pp. 420–435. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-46982-9_26

[30] C. Drabek, T. Pramsohler, M. Zeller, and G. Weiss, "Interface verification using executable reference models: An application in the automotive infotainment." in 6th International Workshop on Model Based Architecting and Construction of Embedded Systems, ACESMB 2013. Proceedings, Miami, Florida, USA, Sep. 2013. [Online]. Available: http://ceur-ws.org/Vol-1084/paper7.pdf