Method for Automatic Resumption of Runtime Verification Monitors

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*—In networked embedded systems created with parts from different suppliers, deviations from the expected communication behavior often cause integration problems. Therefore, runtime verification monitors are used to detect if observed communication behavior fulfills defined correctness properties. However, in order to resume verification if unspecified behavior is observed, the runtime monitor needs a definition of the resumption. Otherwise, further deviations may be overlooked. We present a method for extending state-based runtime monitors with resumption in an automated way. This enables continuous monitoring without interruption. The method may exploit diverse resumption algorithms. In an evaluation, we show how to find the best suited resumption extension for a specific application scenario and compare the algorithms.**

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

I. Introduction

In-car infotainment systems are an example for the increasing complexity of software services in networked embedded systems. Common basic architectures are utilized to enable faster development cycles, reuse, and shared development of non-differentiating functionality. Interoperable standards enable the integration of software components from multiple vendors into one platform. However, the integration of such services remains a challenge, since not only static interfaces have to be compatible but also the interaction behavior.

Even though single functions are tested thoroughly for their compliance to the specification, deviations in the behavior occur often when new functions are integrated into a complete system, e.g., caused by side-effects on timing by other functions, misconfiguration or incomplete specifications. Further, isolated testing is not feasible for all functionality, because of the exhaustive and sometimes unknown test-contexts that would be required. In these situations, it is vital to be able to monitor the interactions of the integrated system at runtime to detect deviations from the expected behavior. Nevertheless, a robust system continues its work after a non-critical failure or deviation from its specification; therefore, its monitors must also be able to resume verification after an observed deviation.

A finite state machine (FSM) can be used to specify the communication behavior in the networked embedded system. Such a reference model can also be generated from observed behavior and is quite versatile. It 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 reference model can be used as a monitor [1]. It is run in parallel to the system under test (SUT) and cross-checks the observed interactions with its own modeled communications (cf. Figure 1). As this

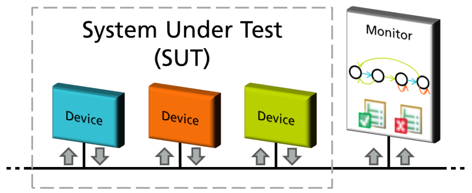


Figure 1. Monitor using a reference model to verify communication behavior.

model is directly derived from the specification, the monitor effectively compares the observation with the specification.

However, specifications often leave room for interpretation, in particular concerning handling of errors. Hence, it is undefined how a monitor based on such a specification should continue after a deviation. The adaptation to make the monitor resilient is usually done manually and needs to be maintained.

To reduce the effort and room for mistakes, we promote using a specification-based monitor and automating the process of making it resilient. We introduce a method that completes the transition function of such a monitor. Thereby, the extended monitor is granted the ability to resume its observation after deviations. The same monitor instance can be used to find multiple deviations. We call this resumption. When using a resumption extension, 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. Moreover, resumption eliminates the need to split the specification into multiple properties. If available, we suggest to use the reference model of the specification as basis for the monitor. Thereby, it is easier to understand deviations, as they can be presented in the context of the whole specification. Further, the reuse of the specification guarantees compliance of the monitor. By exchanging the resumption algorithm ($\mathcal{R}$) generating the extension, the monitor's resilience can be optimized for the current application scenario.

This work introduces the general method of resumption and how resumption algorithms can be evaluated. To demonstrate the evaluation, we also present and compare different algorithms. They recreate patterns that we found to be commonly used when manually improving the resilience of a FSM. The evaluation framework and metrics help to find the best suited extension for individual systems.

The rest of this paper is structured as follows. After discussing related work in Section II, Section III describes the concept of specification-based monitors and the necessary notation. Section IV introduces the method of resumption and the algorithms considered in this paper. In Section V, we present the evaluation and discuss the findings. Section VI concludes the paper and gives an outlook on future work.

II. Related Work

Various areas address the problem of detecting differences between a SUT'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 [2]. It is used offline, i.e., after the SUT finished its execution, because the employed data mining techniques to match model and execution are computationally intensive and can only be used efficiently once the complete logs are available. In contrast, the presented resumption uses assumptions on the expected deviations to provide lean algorithms that work at runtime.

Model-based testing aims to find differences between the behavior of a SUT and a valid behavior model [3]. An environmental [4] or embedded [5] 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. Before each test set, the SUT is actively maneuvered into a known state using a homing sequence. Generally, these sequences reduce the current state uncertainty by utilizing separating or merging sequences [6]. 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 [6]. However, a passive monitor should not influence the SUT. Therefore, the presented resumption cannot actively force the system to a known state. Nevertheless, occurrences of separating and merging sequences can be tracked during observation 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 SUT satisfies or violates a correctness property [7]. The observation of communication is well suited for black box systems, as no details about the inner states of the SUT are needed. Further, the influence on the SUT 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 [8]. Runtime verification frameworks, such as TraceMatches [9] or JavaMOP [10], preprocess and filter the input before it is passed to a monitor instance. Thereby, each monitor only sees relevant events. A property-based monitor checks if a certain subset of the specification is fulfilled or violated. Unless extended with resumption, it will only report a single deviation. Nevertheless, if the properties are carefully chosen, the respective monitor can match an arbitrary slice of the input trace. Then, the monitor is instantiated and matched against different slices of the trace. However, this requires that the complete specification is split into multiple of such properties and implies additional design work. Thereby, or if the properties are extracted by data mining techniques from a running system or traces [11][12], a secondary specification is created that needs to be kept in sync. In contrast, resumption enables the reuse of an available specification by automatically augmenting its robustness for verification.

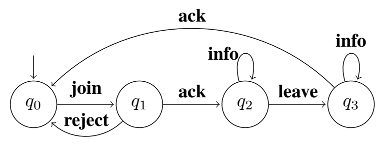


Figure 2. FSM of the communication behavior related to a subscription service.

III. Specification-based Runtime Monitors

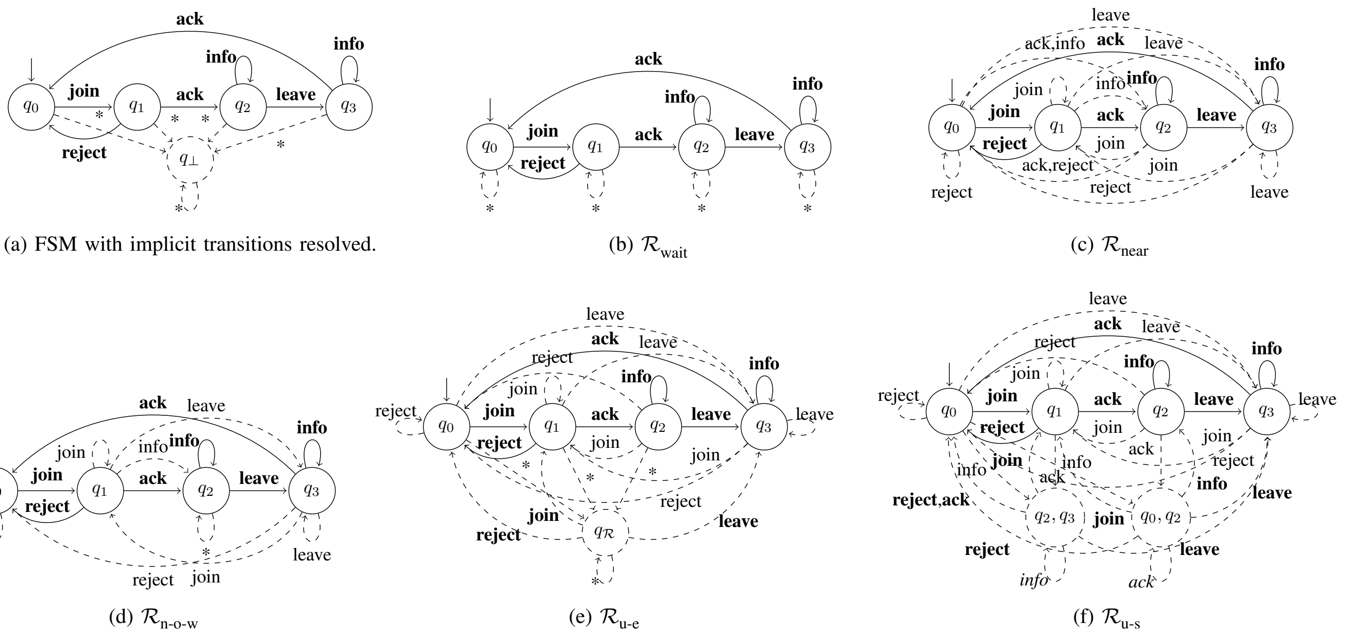
A monitor is "a system that observes and analyses the behavior of another system" [13]. The core of a monitor is an analyzer which is created from the requirements and different languages can be used to specify the analyzer [14], e.g., linear temporal logic [7]. However, without loss of generality, such a description can be mapped to a set of states and a set of transitions between the states [15], i.e., a (finite) state machine.

In diverse embedded system domains like automotive, state machines are often used for the specification of communication protocols or component interactions. We call such a state machine a reference model and a monitor that uses the reference model to check conformance of observed interactions a specification-based monitor. Reference models usually focus on capturing the valid behavior and include only critical or exemplary deviations. Therefore, they only describe a partially defined transition function and a subset of all possible error states. The respective specification-based monitor reports an accepting verdict ($\top$) for valid behavior and a rejecting verdict ($\perp$) or another associated verdict for deviations. The FSM in Figure 2 shows a FSM 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 3a. If an event without transition in the original FSM occurs, $q_\perp$ is entered. However, such a monitor will only detect the first deviation. To overcome this, the next section introduces resumption and how the resolution can be performed to overcome this.

Beforehand, we introduce the necessary notation. A monitor $M : \langle \mathbb{D}, \mathbb{A}, \mathbb{Q}, q_0, \delta, \gamma \rangle$ consists of a verdict domain $\mathbb{D}$, an observation alphabet $\mathbb{A}$, a set of states $\mathbb{Q}$, an initial state $q_0 \in \mathbb{Q}$, a transition function $\delta : \mathbb{A} \times \mathbb{Q} \to \mathbb{Q}$ and a verdict function $\gamma : \mathbb{A} \times \mathbb{Q} \to \mathbb{D}$. For a specification-based monitor, M is identical to the reference model and, thereby, identical to the specification. The observation alphabet $\mathbb{A}$ and the verdict domain $\mathbb{D}$ of the monitor are the input and output sets of the state machine. The latter is a set of verdicts, at least containing $\top$ and $\perp$. The former is a set of semantic events used to distinguish the different interactions of the SUT relevant for the monitor. At runtime, there are various ways to extract the semantic events by preprocessing and slicing the observed interactions, e.g., [9][10][15][1]. In the following, we will refer to them in general as events.

Let $\mathrm{dom}(\delta)$ be the domain of a partial function, such as δ, i.e., the set of elements with a defined mapping. Let $\mathbb{A}^q$ be the set of events with a defined transition in state q (1), $\mathbb{Q}^e$ be the set of states with a defined transition for event e (2) and $\mathbb{Q}^{e,\delta}$ be the set of defined target states for event e (3).

$$\mathbb{A}^q = \{e \in \mathbb{A} \mid \langle e, q \rangle \in \mathrm{dom}(\delta)\} \tag{1}$$



(a) FSM with implicit transitions resolved. (b) $\mathcal{R}_{\text{wait}}$ (c) $\mathcal{R}_{\text{near}}$ (d) $\mathcal{R}_{\text{n-o-w}}$ (e) $\mathcal{R}_{\text{u-e}}$ (f) $\mathcal{R}_{\text{u-s}}$

Figure 3. FSMs with states and transitions (dashed) added by the implicit error assumption (a) and different $\mathcal{R}$ (b)-(f). **Bold** labels indicate an accepting, regular labels a rejecting, and *italic* labels an inconclusive verdict. The wild-card '*' matches all events that have no other transition in the state.

$$\mathbb{Q}^e = \{q \in \mathbb{Q} \mid \langle e, q\rangle \in \text{dom}(\delta)\} \tag{2}$$

$$\mathbb{Q}^{e,\delta} = \{q_t \in \mathbb{Q} \mid \exists q_s \in \mathbb{Q}^e : \delta(e, q_s) \mapsto q_t\} \tag{3}$$

IV. Resumption

A specification-based monitor, such as shown in Figure 3a, will only be able to find the first deviation from the specification, since it enters the final state $q_\perp$ at this point. Different techniques can be applied in order to create resilient 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 additional transitions and triggers or to artificially split the specification into multiple properties that can be checked separately. However, we suggest using a generic definition for how the monitor can resume its duty.

This section presents the method for *resumption* that enables a monitor to analyze the trace for additional deviations with respect to the same property. This can be used to resume the operation of the monitor, e.g., after a deviation was detected or for initialization, and is especially useful when the system under test cannot be forced into a known state.

Example 1 (Resumption): Let's assume M in state q observes event $\chi \in \mathbb{A} \setminus \mathbb{A}^q$, i.e., the specification defines no transition for χ in the active state. By the definition of a specification-based monitor, a deviation is reported. However, as the event is undefined for this state in the specification, additional information is required for the monitor to continue observation. If the application scenario allows to ignore the deviating event, the monitor can stay in the same active state and continue its work.

A. Resumption Extension

Any specification-based monitor may be extended with the help of a resumption extension. Even a monitor that has a complete transition function may have need for resumption, if it has unrecoverable states like $q_\perp$ in Figure 3a. To distinguish between the original monitor, the extension, the extended monitor and their components the sub-scripts $\mathcal{L}$, $\mathcal{R}$ and $\mathcal{E}$ are used respectively. $M_\mathcal{E}$ is created by combining the sets and functions of $M_\mathcal{L}$ with $M_\mathcal{R}$, where $M_\mathcal{L}$ is preferred. However, $\delta_\mathcal{R}$ may override $\delta_\mathcal{R}$ for choosable verdicts, e.g., $\perp$.

Example 2 (Resumption Extension): Figure 3b shows a possible extensions of the FSM given 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 FSM has a complete transition function and can continue to monitor after reporting deviations. Thereby, the original monitor is extended with 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 (4) 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 $\mathbb{Q}_C = \mathbb{Q}_\mathcal{L} \cup \{q_\mathcal{R}\}$ and $\mathcal{P}(\mathbb{Q}_C)$ be the set of all subsets of $\mathbb{Q}_C$.

$$\mathcal{R} : \mathbb{A} \times \mathcal{P}(\mathbb{Q}_C) \to \mathcal{P}(\mathbb{Q}_C) \tag{4}$$

Using $\mathcal{R}$, the additional states and transitions that are needed for the extension of the original monitor can be derived. For finite sets $\mathbb{Q}_\mathcal{L}$ and $\mathbb{A}$, a preparation step creates the states $\mathcal{P}(\mathbb{Q}_C) \setminus \mathbb{Q}_C$. The transitions are derived by evaluating $\mathcal{R}$ to find the target state. If $\mathcal{R}(e, q)$ returns an empty set or solely states that cannot reach any state in $\mathbb{Q}_C$, 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 $\mathbb{Q}_C$ 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 $\mathbb{Q}_{\mathcal{L}}$, $M_{\mathcal{L}}$ can resume verification in that state. Otherwise, the set of candidates is stored and given to $\mathcal{R}$ with the next event.

$\gamma_{\mathcal{R}}$ is defined as follows: it accepts the transitions from $\mathbb{Q}_{\mathcal{R}}$ to $\mathbb{Q}_{\mathcal{L}}$, rejects transitions from $\mathbb{Q}_{\mathcal{L}}$, and returns an inconclusive verdict otherwise. Thereby, the resulting $\gamma_{\mathcal{E}}$ reports the specified verdicts, rejects unexpected deviations and accepts events as soon as it has resumed verification.

B. *Resumption Algorithms*

This section introduces algorithms that can be used for resumption. Often, these algorithms are mimicked to extend specifications manually to create resilient monitors. Based on an observed event and a set of candidates for the active state $\mathcal{R}$ will determine the possible states of the SUT with respect to the observed property. The results of applying the algorithms on the FSM from Figure 2 are shown in Figure 3. 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 look at all states equally.

The local algorithm *Waiting* (5) 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 $\mathbb{A}^q$. $\mathcal{R}_{\text{wait}}$ assumes that a deviation was caused by a superfluous message that may be ignored. It is expected to perform bad for other deviations.

$$\mathcal{R}_{\text{wait}}(e, \mathbb{Q}_{in}) = \mathbb{Q}_{in} \tag{5}$$

An obvious danger is, the SUT may never emit an event that is accepted by the active state. Therefore, the next algorithms also look at states around the active state. The used distance measure $\|q_s, q_t\|$ 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 $\|\mathbb{Q}_s, \mathbb{Q}_t\|$ is the transition count of the shortest path between any state in $\mathbb{Q}_s$ and any state in $\mathbb{Q}_t$.

The algorithm *Nearest* (6) resumes verification with the next event accepted by any state reachable from the active state. If multiple transitions match, it chooses the transition reachable with the fewest steps from the previously active state.

$$\mathcal{R}_{\text{near}}(e, \mathbb{Q}_{in}) = \underset{q_t \in \mathbb{Q}_C^{e,\delta_{\mathcal{L}}}}{\operatorname{argmin}} \min_{q_s \in \mathbb{Q}_{in}} \|q_s, q_t\| \tag{6}$$

$\mathcal{R}_{\text{near}}$ 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.

The algorithm *Nearest-or-Waiting* (7) resumes verification like *Nearest*, except if the selected state is more steps away from the active state than the active state is from any other state that could match the event. 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. $\mathcal{R}_{\text{n-o-w}}$ is a combination of the previous two algorithms and shows how algorithms can be combined to create new ones.

$$\mathcal{R}_{\text{n-o-w}}(e, \mathbb{Q}_{in}) = \begin{cases} \mathcal{R}_{\text{wait}}, & \text{if } \|\mathbb{Q}_C^e, \mathbb{Q}_{in}\| < \|\mathbb{Q}_{in}, \mathcal{R}_{\text{near}}\| \\ \mathcal{R}_{\text{near}}, & \text{otherwise} \end{cases} \tag{7}$$

Global algorithms assume that you need to consider the whole specification to identify the current communication state. Therefore, they look at all states equally to keep all options open for resumption.

Unique-Event (8) 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.

$$\mathcal{R}_{\text{u-e}}(e, \mathbb{Q}_{in}) = \begin{cases} \mathbb{Q}_C^{e,\delta_{\mathcal{L}}}, & \text{if } |\mathbb{Q}_C^{e,\delta_{\mathcal{L}}}| = 1 \\ \{q_{\mathcal{R}}\}, & \text{otherwise} \end{cases} \tag{8}$$

Unique-Sequence (9) 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.

$$\mathcal{R}_{\text{u-s}}(e, \mathbb{Q}_{in}) = \begin{cases} \mathbb{Q}_{in}^{e,\delta_{\mathcal{L}}}, & \text{if } \mathbb{Q}_{in}^{e,\delta_{\mathcal{L}}} \neq \emptyset \\ \mathbb{Q}_C^{e,\delta_{\mathcal{L}}}, & \text{if } \mathbb{Q}_{in}^{e,\delta_{\mathcal{L}}} = \emptyset \wedge \mathbb{Q}_C^{e,\delta_{\mathcal{L}}} \neq \emptyset \\ \{q_{\mathcal{R}}\}, & \text{otherwise} \end{cases} \tag{9}$$

Similar to homing sequences used in model-based testing, $\mathcal{R}_{\text{u-s}}$ aims to reduce the current state uncertainty with each step. In each iteration of the algorithm, 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 SUT emits one, $\mathcal{R}_{\text{u-s}}$ will detect it. Any deviation in the behavior causes $\mathbb{Q}_{in}^{e,\delta_{\mathcal{L}}}$ to be empty and therefore resets the set of possible candidates to any state accepting the event, i.e., the resumption is resumed.

V. EVALUATION

This section presents an evaluation of the introduced method for automatic resumption of runtime verification monitors. Therefore, a framework is employed to compare the presented algorithms.

A. *Evaluation Framework*

An overview of the evaluation setup is depicted in Figure 4. A specific *Application Scenario* usually provides the specification and, thereby, a *Reference Model*. However, to make general statements about the algorithms, a generator creates the models. The resulting FSMs use global events across the whole machine and local groups. To classify the FSMs, 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 FSM that have a unique event.

For each reference model, multiple traces are generated by randomly selecting paths from the respective FSM. The *Deviation Generator* manipulates the FSM used by the trace-generator by adding new states and transitions. These transitions use undefined events ($\chi \notin \mathbb{A}^{q_s}$) of the source state q_s. This guarantees that deviations are detected at this event,

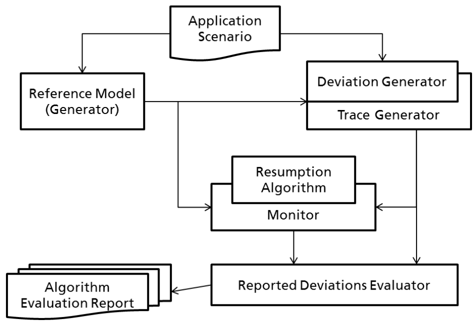

Application Scenario
Reference Model (Generator)
Deviation Generator
Trace Generator
Resumption Algorithm
Monitor
Algorithm Evaluation Report
Reported Deviations Evaluator

Figure 4. Overview of the evaluation framework for resumption algorithms.

if the monitor knows the current state. The added deviations are characterized by the different transition targets q_t: superfluous ($q_t = q_s$), altered ($\exists e : \delta_{\mathcal{L}}(e, q_s) \mapsto q_t$), skipped ($\exists e : \delta_{\mathcal{L}}(e, q_s) \mapsto q_i \wedge \delta_{\mathcal{L}}(\chi, q_i) \mapsto q_t$) and random events ($q_t \in \mathbb{Q}_{\mathcal{L}}$). Additionally, shuffled events are simulated by choosing a chain of two transitions and creating copies in inverse order with a new intermediate state. This is a special case of two altered events in sequence. If a scenario expects more complex deviations, they can be simulated by combining 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 invisible to the monitor.

The traces are eventually checked using the original FSM extended with each $\mathcal{R}$. For the evaluation an Eclipse-based tool capable of using reference models as monitors [1] was used and extended. Using hooks in the tool's model execution runtime, resumption is injected if needed. Thereby, all introduced algorithms can easily be exchanged.

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 detected and the injected deviations. It calculates the well established metrics from information retrieval *precision* and *recall* [16] for each extended monitor. Precision (10) is the fraction of reported deviations (rd) that were true (td), i.e., injected by the deviation generator. Recall (11) is the fraction of injected deviations that were reported. Both values are combined to their harmonic mean, also known as F_1 score (12).

$$p = |td \cap rd| / |rd| \tag{10}$$

$$r = |td \cap rd| / |td| \tag{11}$$

$$F_1 = 2 \cdot \frac{p \cdot r}{p + r} \tag{12}$$

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 loose precision as they attempt to find further deviations. Therefore, recall estimates how likely all true deviations are reported. A regular monitor's recall is $|td|^{-1}$ as it reports only the first deviation.

B. Comparison of Resumption Algorithms

The subscription service example evaluates to the F_1 scores: $\mathcal{R}_{\text{wait}} = 0.53$, $\mathcal{R}_{\text{near}} = 0.68$, $\mathcal{R}_{\text{n-o-w}} = 0.79$, $\mathcal{R}_{\text{u-e}} = 0.80$, $\mathcal{R}_{\text{u-s}} = 0.78$. For the general evaluation, traces with a total of 55 million deviations in 220 different FSMs 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 regular monitor is 0.05 and its F_1 score 0.095. Figure 5 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 deviation but also the best precision for shuffled and skipped deviations. $\mathcal{R}_{\text{u-s}}$ enables better precision for the other deviations, plus a very high recall.

Figure 6 compares the F_1 scores of the algorithms for different levels of uniqueness and numbers of states of the generated FSMs. The low overall score of $\mathcal{R}_{\text{wait}}$ is clearly visible for both metrics. For FSMs with low uniqueness, $\mathcal{R}_{\text{u-s}}$ outperforms the 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, then. 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 $\mathcal{R}_{\text{wait}}$. $\mathcal{R}_{\text{wait}}$ and $\mathcal{R}_{\text{u-s}}$ are hardly affected by the state count.

C. Discussion

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

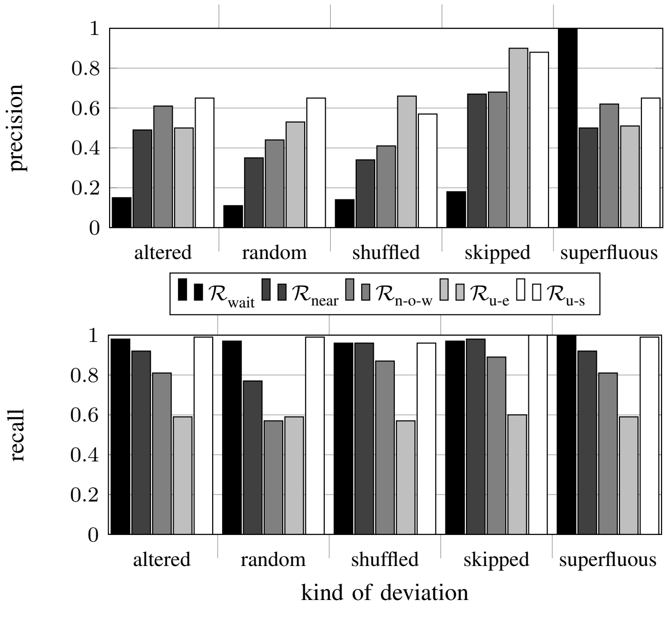


Figure 5. Precision and recall of $\mathcal{R}$ compared for different kinds of deviations.

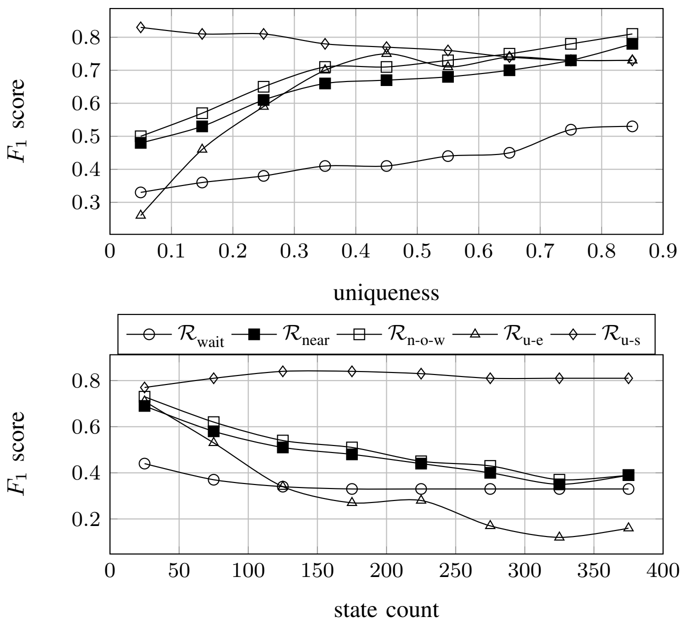


Figure 6. F_1 scores of $\mathcal{R}$ compared for metrics uniqueness and state count.

that knowing which deviations are expected can help formulate specialized algorithms. However, $\mathcal{R}_{\text{wait}}$ performs worst for all other kinds of deviations, as the SUT transitioned internally to a different state already and would have to return to the original state. It benefits from unique events as they prevent taking wrong transitions in the meantime.

The metric uniqueness helps to decide 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 SUT. Of the examined algorithms, only $\mathcal{R}_{\text{u-s}}$ takes multiple events into account and, therefore, should be preferred in this case. 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 an 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, 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 options for which an algorithm may resume and use a local resumption algorithm. 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 $\mathcal{R}_{\text{u-e}}$. Nevertheless, in all cases, the F_1 scores of the extended monitors are always better than what can be calculated for a regular monitor.

The results for the subscription service example (uniqueness 0.43, 4 states) and the respective results from Figure 6 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 an estimation.

VI. CONCLUSION

In this paper, we introduce 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 presented resumption algorithms has its strength and weaknesses. The presented framework and metrics help to find the best suited algorithm for an application scenario. Future work includes improving the method for resumption, e.g., by taking event parameters into account and by handling partially-independent behavior. Moreover, enhanced algorithms that target specific real world scenarios will be examined.

ACKNOWLEDGMENT

The project was funded by the Bavarian Ministry of Economic Affairs, Infrastructure, Transport and Technology.

REFERENCES

[1] C. Drabek, A. Paulic, and G. Weiss, “Reducing the Verification Effort for Interfaces of Automotive Infotainment Software,” SAE Technical Paper 2015-01-0166, 2015.

[2] 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, 2012, pp. 182–192.

[3] A. Pretschner and M. Leucker, “Model-Based Testing A Glossary,” in Model-Based Testing of Reactive Systems. Springer Heidelberg, 2005, pp. 607–609.

[4] 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. LNEE. Springer Singapore, 2016, no. 382, pp. 203–213.

[5] A. Kurtz, B. Bauer, and M. Koeberl, “Software Based Test Automation Approach Using Integrated Signal Simulation,” in SOFTENG 2016, Feb. 2016, pp. 117–122.

[6] S. Sandberg, “Homing and Synchronizing Sequences,” in Model-Based Testing of Reactive Systems. Springer Heidelberg, 2005, pp. 5–33.

[7] M. Leucker and C. Schallhart, “A brief account of runtime verification,” The Journal of Logic and Algebraic Programming, vol. 78, no. 5, May 2009, pp. 293–303.

[8] S. D. Stoller et al., “Runtime Verification with State Estimation,” in Runtime Verification. Springer Berlin Heidelberg, 2012, pp. 193–207.

[9] C. Allan et al., “Adding Trace Matching with Free Variables to AspectJ,” in OOPSLA ’05. ACM, 2005, pp. 345–364.

[10] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Rou, “An overview of the MOP runtime verification framework,” Int J Software Tools Technology Transfer, vol. 14, no. 3, Apr. 2011, pp. 249–289.

[11] A. Danese, T. Ghasempouri, and G. Pravadelli, “Automatic Extraction of Assertions from Execution Traces of Behavioural Models,” in DATE ’15, 2015, pp. 67–72.

[12] F. Langer and E. Oswald, “Using Reference Traces for Validation of Communication in Embedded Systems,” in ICONS 2014, pp. 203–208.

[13] D. K. Peters, “Automated Testing of Real-Time Systems,” Proc. Newfoundland Electrical and Computer Engineering Conference, 1999.

[14] N. Delgado, A. Gates, and S. Roach, “A taxonomy and catalog of runtime software-fault monitoring tools,” IEEE Transactions on Software Engineering, vol. 30, 2004, pp. 859–872.

[15] Y. Falcone, K. Havelund, and G. Reger, “A Tutorial on Runtime Verification.” Engineering Dependable Software Systems, vol. 34, 2013, pp. 141–175.

[16] 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.