Kalpa Publications in Computing
Volume 3, 2017, Pages 71–80
RV-CuBES 2017. An International Workshop on
Competitions, Usability, Benchmarks, Evaluation,
and Standardisation for Runtime Verification Tools

DANA - Description and Analysis of Networked Applications

Christian Drabek[1] and Gereon Weiss[2]

[1] Fraunhofer ESK, Munich, Germany
christian.drabek@esk.fraunhofer.de
[2] Fraunhofer ESK, Munich, Germany
gereon.weiss@esk.fraunhofer.de

Abstract

We introduce the DANA platform for specifying and analyzing networked applications. DANA was originally created targeting the automotive domain for the verification and validation of software interface behavior in new infotainment and advanced driver assistant systems that are integrated on a single hardware platform. The messages in these interfaces can contain complex data, e.g., playlists with images. Therefore, valid behavior is described as a layered reference model. The platform can use the model to generate test cases, code for simulation, and to verify a live or recorded trace. Exchangeable resumption algorithms enable DANA to resume runtime verification after a deviation using the original state machine without manual changes. A generic input model allows quick integration of new sources for messages. Therefore, DANA can easily be applied to other domains where interactive behavior can be observed. In this paper, we present the tool, its layered reference model, and show its application for runtime verification.

1 Introduction

Automotive 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.

Model-based techniques used during the design and integration phase of new automotive infotainment applications play also a major role in the process of validation and verification [12]. In this paper, we introduce the DANA platform[1], an open and modular environment based on Eclipse for specifying and analyzing networked applications. In addition to monitoring, it supports various transformations of its behavior models, e.g., for generating test cases or code for running simulations. Static analyses are available to check conformance to modeling guidelines, metrics for interfaces and compatibility of behavior models. It was originally created to face the various challenges [6] in the infotainment domain:

[1] An evaluation version and more information is available at http://s.fhg.de/DANA

1. **Evolving technologies**: A plethora of middlewares and hardware interfaces are used and regularly replaced. The modeling approach should separate the components' interface behavior and abstract from these technical details.

2. **State explosion**: Manifold software-based features are assembled and executed in parallel with partially interdependent interfaces. The many potential system states and interactions cannot be modeled individually.

3. **Non-functional requirements**: Infotainment is not a safety-critical domain, but the meeting of timing requirements is essential for orchestrating interactive components.

4. **Deviation detection**: The verification should reliably detect an abnormal operation and encircle the actual fault leading to the erroneous functionality.

5. **Incomplete specifications**: Specification models can be incomplete and only define important parts of a system's behavior. A verification mechanism must be able to resume verification after detecting the presence of unexpected behavior.

6. **Distributed development**: The specification may be interpreted by multiple vendors. To ensure compatibility, it should not be misinterpreted. For model-based specifications this can be achieved with executable semantics.

Within this paper, we focus on DANA's approach for model-based runtime verification of communication behavior in distributed systems considering the above characteristics. We present a layered reference model which specifies interactions between components at syntactical and behavioral level. Further, we show how DANA uses such models for runtime verification.

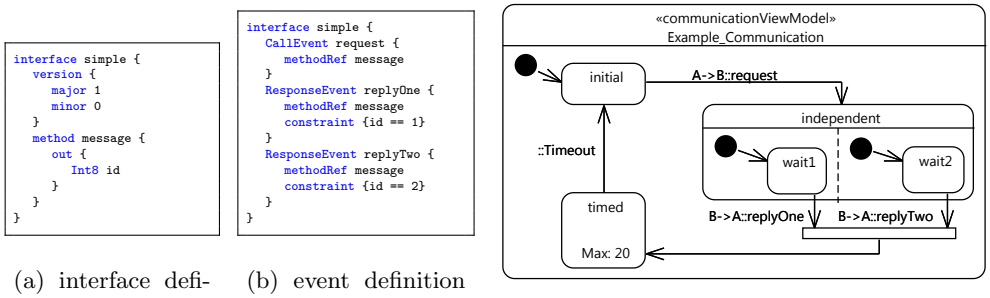
The remainder of this paper is organized as follows: Section 2 introduces layered reference models. We describe the method of resumption in Section 3. Section 4 presents DANA's runtime verification framework. In Section 5, the usage of the platform is demonstrated in various case studies and discussed. Section 6 compares DANA to related work. We conclude the paper with future work in Section 7.

2 Layered Reference Model

Several challenges of interface verification in automotive infotainment systems are addressed by the modeling approach. A so-called *layered reference model* is derived from the requirements. We introduced the main concepts in [6] and [12], and provide an updated description in the following. A layered reference model comprises itself several models and describes the communication between two or more components including all involved interfaces and interactions.

The *interface definition model* describes the artifacts (e.g., broadcasts, methods, and types) of the involved interfaces on a syntactical level. This layer uses Franca IDL [2] as generic description. The specifics for each employed middleware technology can be recorded in additional deployment models. Thereby, the messages can be distinguished and their parameters become accessible. Figure 1a shows a simple interface definition model with a single method.

The *event definition model* refines interface artifacts by listing associated events. Each event consists of a label and a constraint. They describe the parameter values that are considered semantically equivalent. A constraint is defined by basic arithmetic and logical expressions using the parameters of the interface artifact. In turn, the behavioral model can use the simple events and ignore the possibly complex parameters. An example of our textual language for this task is shown in Figure 1b.



(a) interface definition model (b) event definition model (c) behavioral model

Figure 1: Example showing the parts of a layered reference model.

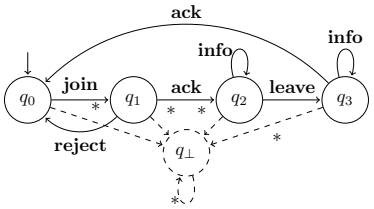
Stereotyped *UML state machines* [11] are used to specify the behavior, i.e., the possible sequences of events in a communication relation. State machines are currently widely applied for behavioral modeling in various domains, e.g., for automotive systems. A UML Profile restricts the variety of UML without limiting its expressiveness. In UML state machines transitions are annotated with triggers and guards. In our experience with real models, the guard conditions are a powerful concept but might become complex pieces of program code. Therefore, we allow the guard condition to be specified using the constraints of the event model only. The sender and receiver of an event are also annotated to the transition. Restricting modeling concepts and removing nested code snippets from the behavior description prevents the use of ambiguous concepts and facilitates a precise description of the communication relationships. Figure 1c shows an exemplary behavioral model for the communication between two components.

A state in our model does not denote the status of single components, but rather represents their common communication state. A state can be annotated with a timeout event which defines the maximum duration the state should be active. When the state has been active for the specified amount of time, the corresponding timeout event is emitted. Triggers may reference an event of the event model or a timeout of the behavioral model. This allows specifying timing requirements in the model (i.e. deadlines and cyclic events).

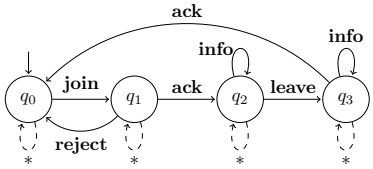
The approach is formalized sufficiently for direct code generation of an executable statechart for any specific middleware. The model allows describing timing as well as parallel and partially interdependent communication. Parallel behavior is modeled by states with parallel regions. A join element is utilized to coordinate the exit of those regions.

3 Resumption

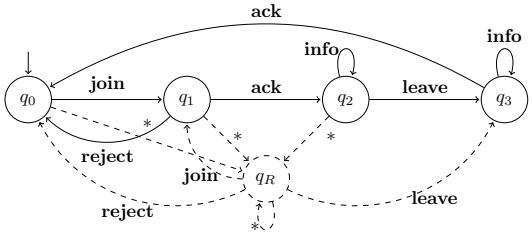
A simple monitor only reports if a given trace conforms to a specification or not. However, it is often of interest to identify every deviation individually. Different techniques can be applied in order to create monitors and to find deviations beyond the first. Up to now, this is usually done manually and requires additional design work, e.g., to add more transitions and triggers, or to split the specification into multiple properties that can be checked separately. With DANA, these limitations are resolved. DANA supports generic definitions for how a monitor can resume its duty. We call this *resumption* [7]. Resumption enables a monitor to analyze a trace for all deviations with respect to the same property. The monitor can resume its operation from an unknown state, e.g., after a deviation was detected or for initialization. This is especially useful if the system under test cannot be forced into a known state.



(a) FSM without resumption.



(b) FSM with *waiting* resumption.



(c) FSM with *unique-event* resumption.

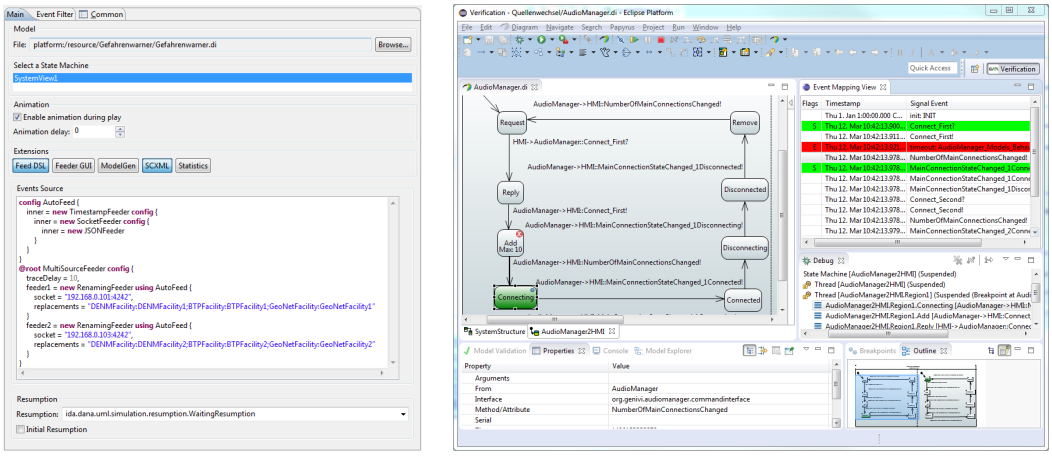
Figure 2: FSMs with states and transitions (dashed) added by the implicit error assumption (a) and resumption algorithms (b)(c). **Bold** labels indicate an accepting verdict. The wild-card '*' matches all events that have no other transition in the state.

Let's assume we use a positive specification as basis for a monitor. Now, if the monitor observes a violation, the specification defines no transition for this event in the active state. An automaton of a monitor may look like Figure 2a. However, this checks only the overall conformance to the specification. Additional work is required for the monitor to report all violations. Generally, a resumption extension completes the transition function of the automation. We suggest to define this extension using a formula that maps a set of candidate states and an event to a set of new candidates for the active state. For example, if the application scenario allows to ignore the deviating event, the monitor can *wait* in the same active state and continue its work. This extension is demonstrated in Figure 2b and defined in (1). Figure 2c shows the extended monitor if *unique events*, i.e., events that always lead to the same state in the specification, are used for resumption (2). $\delta_{\mathcal{L}}$ is the transition function of the specification extended to return the set of states reachable from any of the input states. $\mathbb{S}_C$ is the set of all states including a resumption state q_R. Various algorithms have been compared using the DANA platform in [7]. If (and only if) deviations never influence the current state, ignoring them works perfectly fine. In general, algorithms that identify paths are more reliable, e.g., (3), but handling multiple active state candidates during resumption incurs a certain runtime overhead.

$$R_{\text{wait}}(\mathbb{S}_{in}, e) = \mathbb{S}_{in} \tag{1}$$

$$R_{\text{u-e}}(\mathbb{S}_{in}, e) = \begin{cases} \delta_{\mathcal{L}}(\mathbb{S}_C, e), & \text{if } q_R \in \mathbb{S}_{in} \wedge |\delta_{\mathcal{L}}(\mathbb{S}_C, e)| = 1 \\ \{q_R\}, & \text{otherwise} \end{cases} \tag{2}$$

$$R_{\text{e-b}}(\mathbb{S}_{in}, e) = \begin{cases} \delta_{\mathcal{L}}(\mathbb{S}_{in}, e), & \text{if } \delta_{\mathcal{L}}(\mathbb{S}_{in}, e) \neq \emptyset \\ \mathbb{S}_C, & \text{otherwise} \end{cases} \tag{3}$$



(a) DANA's launch configuration.

(b) DANA animating the results.

Figure 3: Screenshots of DANA being used for runtime verification.

4 Architecture for Runtime Verification

This section describes the platform's architecture for runtime verification [5][6]. It is seamlessly integrated into the Eclipse debugging framework. An Eclipse debug configuration is used to specify the state machine for verification, the source of events, the resumption module and other extensions (cf. Figure 3a). The modules that provide the communication stream can be cascaded and configured using a textual description. Support for new communication media can be provided by implementing an API to access the messages and their properties. For example, we use the available D-Bus bindings for Java [9] to receive messages from the D-Bus of a tested system. The messages are mapped to events using the interface and event definition model. The behavioral model is used as core of the passive monitor for the resulting stream of events. We use SCXML [16] as the execution semantics for the behavior model [6]. If no transition that is reachable from an active state includes a trigger for an observed event, a deviation has been found and is reported. For initialization and after a failure, the monitor resumes operation with the help of a *resumption extension*. If the time stamp of a new event indicates that a state remained active for a longer period than specified in its max-property, a timeout event is injected. The timeout event is otherwise treated like any other event for a communication message. This approach allows the detection of various kinds of failures, e.g., missing messages, additional messages, malformed messages, and timing violations. DANA produces a queue of verdicts that can be presented to the user. Figure 4 shows an overview of the execution framework for reference models used for verification.

For interactive verification, Eclipse will switch to a verification perspective that should be familiar to anyone who already used Eclipse for debugging (cf. Figure 3b). However, a state machine is executed and animated instead of running code. The animation highlights active states and transitions used to enter them. It can pause on found deviations, breakpoints, or the press of a button. While suspended, the stack trace shows the history of states passed. The analysis of new events continues in the background. The queue concept enables to slow down the animation, so that events received in quick succession are still visually observable. Moreover, the animation can replay any subsequence of the queue. Found deviations are marked in the state machine and are listed in the problem view of Eclipse.

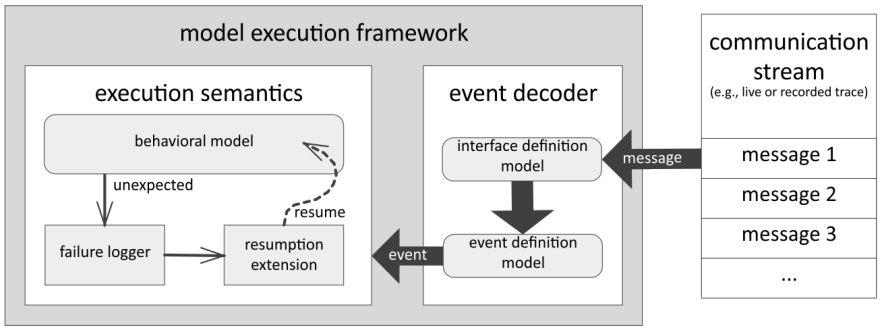


Figure 4: Execution framework for reference models.

5 Use-Cases

In this section, we provide examples of applying DANA in several use cases. First, we present an example for verifying the trace of a *parking assistance service (ParkA)* with *waiting* resumption [6]. Figure 5a shows the interface description model of the ParkA component, Figure 5b depicts the respective event definition model for the ParkA interface. Figure 5c shows the behavioral model for a specification-conformant interaction with a ParkA service, next to an example for a communication trace. These descriptions are used to decode the raw observations into events. Figure 5c shows the behavioral model for a specification-conformant interaction with a ParkA service and an example for an decoded communication trace. Each line lists an identifier, a time stamp, and the event's name. For illustration purposes, the order in which the trace is executed is annotated next to the transitions of the behavioral model. This corresponds to how DANA's animation would highlight the transitions and target states.

In another use case, we successfully employ DANA to support the development of a *hazard warning application* [15]. A sudden obstacle in traffic can be dangerous. Especially, if drivers realize the obstacle too late. A hazard warning can help inform drivers in time. For example, in Figure 6a the driver in the car on the left notices an obstacle and brakes hard. As the view of right car's driver is occluded by the van in the middle, she would only be able to notice this by 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 to be considered. DANA is also capable to address such distributed networked systems, as connected cars. We can capture the to-be-verified-behavior of all involved cars in a model and use this for verification. Figure 6b shows a behavior model for this use case. Hooks in the used communication stack for car-to-car communication are employed to monitor the different communication layers. On the left side it tracks the communication stack of the sending car from detecting the event until the radio waves are emitted to the surrounding. The middle part of the depicted behavioral model comprises the handling of this hazard warning by the receiving car. The right hand side of the figure encompasses a timeout for the duration from detecting the obstacle to triggering a warning in the receiving car. By using this model for runtime verification in DANA, deviations in the hazard warning application 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 statechart.

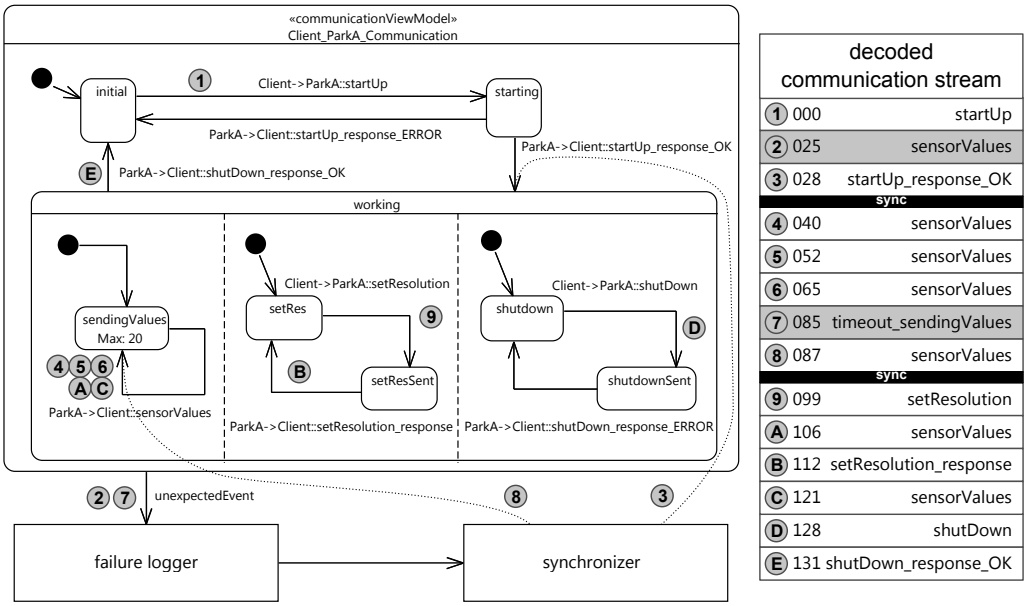
Besides the automotive domain, the DANA platform can also be employed for other application scenarios. Figure 7a shows the controllers of a small industrial plant composed of three stations. The plant assembles cubes from two halves. The first station collects parts from two

```
interface ParkA{
  method startUp{
    in{Resolution res}
    out{AM_Error returnValue} }
  broadcast sensorValues {
    out{
      Int32[] front_lmr
      Int32[] rear_lmr
      Resolution res } }
  enumeration Resolution{ cm mm }
  enumeration AM_Error{ OK ERROR }
  ... }
```

(a) ParkA interface definition excerpt.

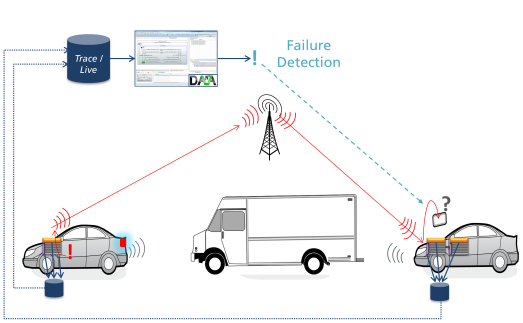
```
Interface ParkA {
  CallEvent startUp {
    methodRef startUp }
  ResponseEvent startUp_response_OK {
    methodRef startUp
    constraint {returnValue == OK} }
  ResponseEvent startUp_response_ERROR {
    methodRef startUp
    constraint {returnValue != OK} }
  BroadcastEvent sensorValues {
    methodRef sensorValues }
  ... }
```

(b) ParkA event definition excerpt.

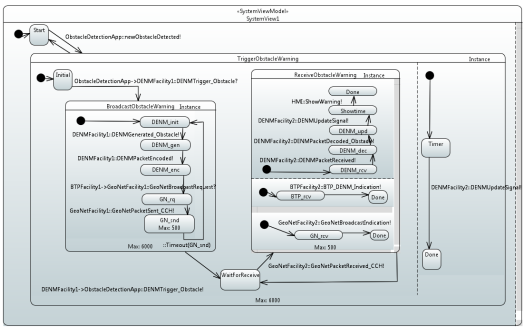


(c) ParkA behavioral model and example trace.

Figure 5: Reference model for the communication between the ParkA component and a client. Additionally the decoded input communication stream is shown.

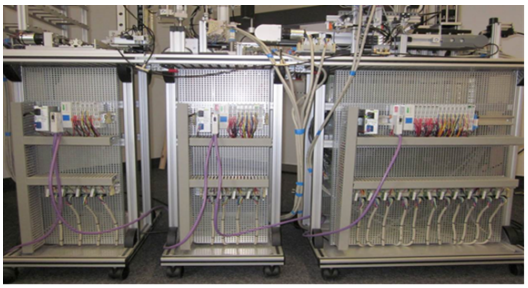


(a) Illustration of the use case.

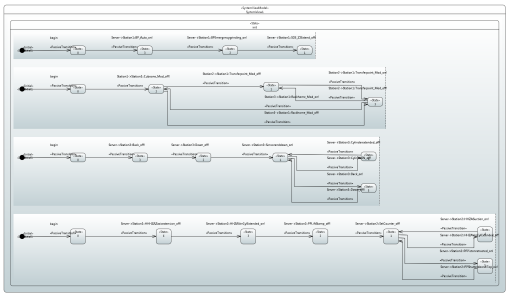


(b) Behavioral model for the use case.

Figure 6: Use case of a hazard warning application.



(a) A small industrial plant with 3 stations.



(b) Behavioral model for the use case.

Figure 7: Use case of a small industrial plant.

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. Even though this information is not needed for the operation of the plant, it enables monitoring the plant's operation without having to alter the original control program. Further, the communication between the stations is recorded by tapping into the switch of the Ethernet-based Modbus TCP connection. Therefore, the behavioral model shown in Figure 7b consists of four parallel regions. Three regions represent the top level behavior of the sensors and actors of each station, and the fourth region includes the communication interaction between them. Actually, the model contains much more details as it was automatically learned from observed behavior, i.e., each of the states contains sub-states that 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.

The case-studies illustrate how DANA addresses the challenges for verifying automotive infotainment interfaces introduced in Section 1 and how it enables the verification of many event based systems:

1. The interface and event definition models abstract from technical details; thus, allowing for a clear behavioral model.

2. All potential system states of parallel and partially interdependent components can be captured by using parallel regions and multiple interfaces in the reference model.

3. The verification mechanism allows to detect timeouts during the verification, e.g., if a message is not delivered in time, by treating timeouts of states like other events.

4. Reliable error detection is provided by logging deviations, which can later be evaluated by the developer, in order to improve the implementation or the specification.

5. Using resumption, the verification framework can resume its operation without manually specifying triggers for this purpose. This helps during initialization and after deviations, i.e., when the state of the observed system is unknown.

6. The mapping of the layered reference model to SCXML provides a clear execution semantic and gives a precise definition of the expected communication behavior.

DANA was originally designed for verifying infotainment components' interfaces by detecting deviations within the communication behavior. However, the tool and concept of layered reference models can be applied to any system which has states and provides observable events that can be used to identify transitions.

6 Related Work

Various areas address the problem of detecting differences between a system's behavior and its specification model. However, the authors are unaware of other approaches that provides similar efficient resumption at runtime. *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 [14]. Cook et al. [4] 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. Reger [13] suggests to include the origin of an event into the analysis to find sensible edits that are consistent for the same origin and correct the trace. Nevertheless, the computational complexity to find an edit sequence requires this to be done offline. In contrast, resumption does not yield precise edits, but can be performed online, in parallel to the execution of the SUO.

Runtime verification frameworks, such as TraceMatches [1] or JavaMOP [10], preprocess and filter the input before it is passed to a monitor instance. Thereby, each monitor only observes relevant events. These stages utilize the first two layers of the layered reference model. Simply keeping the monitor running after it encountered and reported a violation only works in very specific scenarios. Nevertheless, if the properties are carefully chosen, multiple instances of the monitor can match different slices of an input trace [1][3]. However, this requires a secondary specification that needs to be maintained. In contrast, the presented resumption enables the reuse of an available specification by automatically augmenting it for continuous verification.

JavaMOP [10] allows to utilize exchangeable logic plugins to monitor behavior. BeepBeep [8] composes several processors in a *pipe* using a query language to analyze complex events. Similar, DANA utilizes plug-able modules to transform an arbitrary input to a sequence of events and process them. Often needed setups like monitoring using a state machine or generating a model are bundled into configurable extensions (cf. Figure 3a).

7 Future Work

In this paper, we have presented the modular platform DANA. While DANA was originally designed for the verification of automotive infotainment systems, it can be easily adapted to a wide variety of use cases. The layered reference models provide a versatile and precise, yet clear way to describe interfaces of software components and their expected behavior. The runtime verification framework can directly use this description to find all observable deviations in the systems behavior, by employing the concept of resumption. Future work includes the exploration of new applications for the platform and prototyping new algorithms which can be applied during runtime verification, e.g., the collection of runtime statistics that need the current state of the system. Further, we will investigate how layered reference models created by machine learning algorithms can be employed to run regression tests or to predict failures efficiently.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondej Lhotk, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. 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*, OOPSLA '05, pages 345–364, New York, NY, USA, 2005. ACM.

[2] Klaus Birken. Franca - A framework for defining and transforming interfaces. Web: http://franca.github.io/franca/, September 2017.

[3] Mikhail Chupilko and Alexander Kamkin. Runtime Verification Based on Executable Models: On-the-Fly Matching of Timed Traces. *EPTCS*, 111,:2013,pp.67–81, March 2013.

[4] Jonathan E. Cook, Cha He, and Changjun Ma. Measuring behavioral correspondence to a timed concurrent model. In *IEEE International Conference on Software Maintenance, 2001. Proceedings*, pages 332–341, Washington, DC, USA, 2001. IEEE Computer Society.

[5] Christian Drabek, Annette Paulic, and Gereon Weiss. Reducing the Verification Effort for Interfaces of Automotive Infotainment Software. SAE Technical Paper 2015-01-0166, 2015.

[6] Christian Drabek, Thomas Pramsohler, Marc Zeller, and Gereon Weiss. Interface Verification Using Executable Reference Models: An Application in the Automotive Infotainment. In *Proceedings of the 6th International Workshop on Model Based Architecting and Construction of Embedded Systems, ACESMB 2013*, Miami, Florida, USA, 2013. CEUR-WS.

[7] Christian Drabek, Gereon Weiss, and Bernhard Bauer. Method for Automatic Resumption of Runtime Verification Monitors. In *SOFTENG 2017, The Third International Conference on Advances and Trends in Software Engineering*, pages 31–36, Venice, Italy, April 2017. ThinkMind.

[8] Sylvain Hall. When RV Meets CEP. In *Proc. RV 2016*, pages 68–91. Springer, September 2016.

[9] Matthew Johnson. Java D-Bus. https://dbus.freedesktop.org/doc/dbus-java/, Sep. 2017.

[10] Patrick ONeil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rou. An overview of the MOP runtime verification framework. *International Journal on Software Tools for Technology Transfer*, 14(3):249–289, April 2011.

[11] Object Management Group (OMG). Unified Modeling Language Specification Ver. 2.5. OMG Document Number formal/15-03-01 (http://www.omg.org/spec/UML/2.5/), 2015.

[12] Thomas Pramsohler, Mahmut Kafkas, Annette Paulic, Marc Zeller, and Uwe Baumgarten. Control Flow Analysis of Automotive Software Components Using Model-Based Specifications of Dynamic Behavior. *SAE Int. J. Passeng. Cars - Electron. Electr. Syst.*, 6:425–436, April 2013.

[13] Giles Reger. Suggesting edits to explain failing traces. In *Proc. RV 2015*, pages 287–293. Springer, 2015.

[14] Wil van der Aalst, Arya Adriansyah, and Boudewijn van Dongen. Replaying history on process models for conformance checking and performance analysis. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(2):182–192, 2012.

[15] Gereon Weiss and Josef Jiru. Software implementieren und absichern - Mit Modellierung zum schnelleren Prototyping. *Embedded Design*, 3:42–44, 2017.

[16] World Wide Web Consortium (W3C). State Chart XML (SCXML): State Machine Notation for Control Abstraction. W3C Recommendation 15-09-01 (https://www.w3.org/TR/scxml/), 2015.