Architectural Runtime Verification

Lars Stockmann
Software Engineering Group
Heinz Nixdorf Institute, Paderborn University
Paderborn, Germany
lars.stockmann@hni.uni-paderborn.de

Sven Laux
dSPACE GmbH
Paderborn, Germany
slaux@dspace.de

Eric Bodden
Software Engineering Group
Heinz Nixdorf Institute, Paderborn University
Paderborn, Germany
eric.bodden@uni-paderborn.de

***Abstract*—Analyzing runtime behavior is an important part of developing and verifying software systems. This is especially true for complex component-based systems used in the vehicle industry. Here, locating the actual cause of (mis-)behavior can be time-consuming, because the analysis is usually not performed on the architecture level, where the system has initially been designed. Instead, it often relies on source code debugging or visualizing signals and events. The results must then be correlated to what is expected regarding the architecture. With an ever-growing complexity of the systems, the advent of model-based development, code generators and the distributed nature of the development process, this becomes increasingly difficult.**

This paper therefore presents Architectural Runtime Verification (ARV), a generic approach to analyze system behavior on architecture level using the principles of Runtime Verification. It relies on the architecture description and on the runtime information that is collected in simulation-based tests. This allows an analyst to easily verify or refute hypotheses about system behavior regarding the interaction of components, without the need to inspect the source code.

We have instantiated ARV as a framework that allows a client to make queries about architectural elements using a timed LTL-based constraint language. From this, ARV generates a Runtime Verification monitor and applies it to runtime data stored in a database.

We demonstrate the applicability of this approach with a running example from the automotive industry.

***Index Terms*—Architecture, Runtime Verification, Database, AUTOSAR.**

I. Introduction

Verification of control devices is an important aspect in many industries. Especially in the transportation industry, safety requirements and a highly distributed development process command manufacturers and suppliers to perform extensive testing (e.g., the ISO 26262 standard [1]). The software of such devices is usually developed in a component-based fashion, where each software component is first implemented and tested individually before being integrated.

The verification of the integrated systems is difficult, as both software architectures and hardware setups have become more distributed and heterogeneous. Methods such as model checking, which might be applicable to individual components (or parts thereof), suffer from the state-explosion problem and cannot be used on integration level. Also, system specifications are often informally stated [2] prohibiting formal methods. Thus, the main verification method at integration level is simulation-based testing.

Whenever a test fails, developers are tasked with uncovering the actual cause for the misbehavior. Given the complexity of an integrated system, this can be difficult. The actual defect is often not found where a violation or an undesired effect is observed. Thus, to find the root cause efficiently, the system's behavior as a whole must be taken into account. A common systematic approach is to induce hypotheses about a possible cause from the observed behavior and to verify or refute these hypotheses by testing predictions of the behavior. This is commonly referred to as the *scientific method* [3].

Traditional ways of testing hypotheses about the behavior of an integrated system are signal plotting and source code analysis, e.g., using step-by-step debugging. However, a developer already must have some idea about the problem to e.g., find a good spot for a break point. In case of a real-time simulation, source code debugging is often not even an option. Also, it is not feasible to plot and analyze every individual signal. Selecting a new signal that has not been recorded previously requires to rerun the test. This can make the whole process time-consuming [4].

This paper therefore presents Architectural Runtime Verification (ARV), which enables subsequent runtime analysis on architecture level. This means, a user can verify/refute hypotheses after the testing is over to find the root cause for its failure. ARV uses the ideas of Runtime Verification–a technique that in the past has been used mostly to verify the behavior of software components. ARV targets the integration rather than the individual components. The goal is to give a better understanding of the system behavior as a whole. Also, it can serve as a starting point for the aforementioned traditional approaches, because a developer can infer from a verified hypothesis on architecture level, where to set a break point in the code or which signals might be worth analyzing. To summarize, this paper presents the following original contributions:

1) a requirements analysis for ARV, i.e., a framework that allows Runtime Verification to be applied on the architecture level (Section IV)

2) a domain-independent model for ARV that augments the structure of a generic architecture with runtime information (Section V-A and Section V-B)
3) a way to formally specify hypotheses about the runtime that can be verified by ARV (Section V-C)
4) a way to verify the hypotheses using a Runtime Verification monitor applied to a database, without having to query every single event (Section V-D)

It is further structured as follows: The next section first explains the main concepts of Runtime Verification. Section III introduces an illustrating running example that has also been used for the evaluation. After that, Section IV presents a requirements analysis that forms the basis for the actual approach presented in Section V. Section VI describes how we evaluated ARV and Section VII presents related work. The paper concludes with a brief outlook in Section VIII.

II. Runtime Verification and Linear Temporal Logic

Falcone et al. defined Runtime Verification (RV) as a "dynamic analysis method aiming at checking whether a run of the system under scrutiny satisfies a given correctness property" [5]. This means that compared to model checking, it focuses on a concrete execution of a system rather than all possible execution paths. It typically works by first translating a given property into a monitor (synthesis) that observes the system's execution in a second step called monitoring. Here, the monitor consumes events produced by the system and outputs a verdict reflecting the satisfaction of the property.

In many approaches, the monitoring is performed at runtime, i.e., it becomes a part of the system. This allows to employ finite automata for the actual verification, which does not require storing runtime data. The properties are often specified using some variant of Linear Temporal Logic (LTL) from which the automata-based monitor is synthesized. LTL is a formalism presented by Pnueli in [6] that treats time in a linear way, i.e., for every state of the system there is exactly one next state.

A more comprehensive introduction of LTL can be found in [2]. RV is well presented in the surveys by Leucker et Schallhart [7] and Falcone et al. [5]

III. Running example

This section presents a simple example architecture of an automotive indicator system that we use to explain the approach. It can be found in the SystemDesk[1] modeling tutorial [8] and is depicted in Fig. 1. It is particularly useful to illustrate the main concepts of *AUTOSAR*[2], which offers a standardized component-based architecture description that involves different layers [9]. For simplicity reasons, it only includes the front left and front right direction indicators. The general functionality of such a system should be familiar. The front left/right indicator is triggered by an indicator switch that can be pushed down or pulled up. A hazard warning light switch triggers both direction indicators simultaneously. Corresponding sensors measure the current status of the switches and actuators are used to turn on/off the lights of the direction indicators. Assume that there is a failed test which involves the hazard warning light switch. The following exemplary hypotheses could be interesting to get a better understanding what goes wrong:

H1 *The runnable of the IndicatorLogic component is invoked (anytime).*
H2 *The warning light switch is activated (anytime).*
H3 *The indicators are always activated within 200ms after the warning light switch has been pressed.*

IV. Requirements Analysis

ARV's concept is based on a requirements analysis that we conducted in advance. This section gives an overview of the main goals that we persued with ARV's design.

A. The Architecture Model

One requirement is that ARV should be usable beyond domain boundaries. This is because simulation systems are often composed of components from different domains, which themselves might be architectures. For example, on the highest level, there are components representing the controller (the system under test) and some representing the environment. The latter is usually composed out of single-component models, e.g., Functional Mockup Units (FMUs [10]). In the automotive industry, the controller can be an AUTOSAR-based, which means that it forms an architecture on its own. AUTOSAR features some unique architectural elements like "Runnable Entities" that wrap the behavioral code and are mapped to tasks of the operating system. Other domains can have custom descriptions that are created using, e.g., the Architecture Analysis & Design Language (AADL) [11]. From this it follows that ARV's underlying architecture model must be very generic with the capability to represent hierarchies.

B. Recording Runtime Data

Another goal of an ARV framework is to allow testers and integrators to verify/refute hypotheses about the runtime of an architecture without having to repeat a test. Therefore, the runtime information must be recorded. This has been done since the beginning of simulation-based testing of electronic control devices. From an engineer's view, runtime information mainly comprises signal curves. Looking from the software perspective, these signals are just observed changes in variables over time. It is worth noting that at integration level, this implies an event-discrete nature of the system under scrutiny. Thus, every variable change can be seen as an event. Every such event has a timestamp, which

[1]SystemDesk: www.dspace.com/go/systemdesk
[2]AUTomotive Open System ARchitecture: www.autosar.org

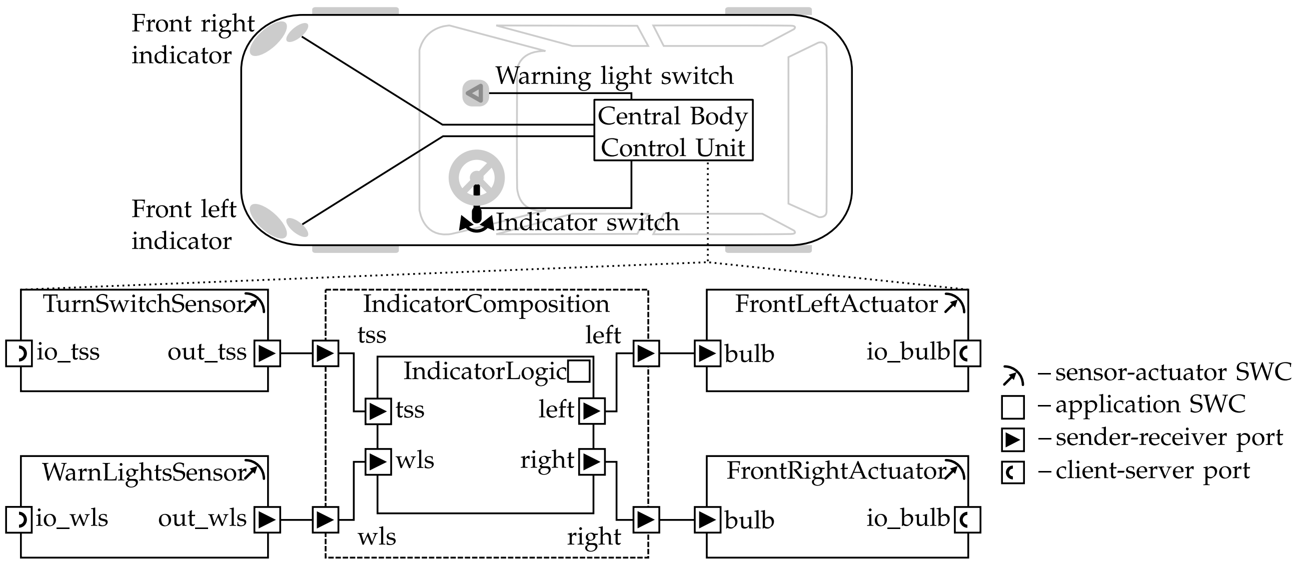

Fig. 1. Simple AUTOSAR Indicator Example

implies a temporal relation to other events. However, depending on the resolution of the measurement, a timestamp is not necessarily unique. This means that the order in which events are recorded must be preserved.

In the past, only signals that were deemed significant for later analysis were recorded—often in a limited time frame to reduce the amount of generated data. A complete analysis requires all runtime data. It can be argued that even with today's high capacity storage solutions, it would be infeasible to log every instruction and all changes in every local variable for hundreds of different configurations and tests. Fortunately, ARV only requires runtime information related to the architecture, which reduces the amount of data significantly.

C. Formulating Hypotheses

In order to verify/refute hypotheses, the integrator has to formulate them first based on the architectural elements. According to the RV mechanism, the hypotheses must then be translated into formal correctness properties, which can be evaluated automatically using the RV monitor. Two main aspects must be considered here: First, it must be possible to specify simple statements/assertions about the current state (e.g., as in H2 of the running example that the WarnLightsSensor's input value is 1) or events that evaluate to either true or false (e.g., as in H1 that a runnable is invoked). These atomic propositions form the building blocks for more complex queries, where they are combined using logical operators. However, it is insufficient to just consider values of ports. The integrator wants to know, when a component was active, i.e., when its code was executed. This is because in the integration phase a component not executing when anticipated, or doing so unexpectedly is a common problem. Even though the actual component behavior is not formally specified in the architecture, its execution can be inferred from the runtime information. Second, it must be possible to specify temporal relations between states or events, i.e., if a proposition is true a certain number of milliseconds before or after another proposition. This is very important, because many behavioral anomalies on integration level are due to an unexpected order in which components execute. In RV, it is common to use LTL (or a timed variant) for the specification [12], [13].

D. Accessing the Data

The framework must ensure efficient queries, i.e., verifying a hypothesis should not require a client to download and process all runtime data. Engineers increasingly expect to be able to work with tablets or even smartphones and use browser-based applications. At the same time, databases are usually backed by high performance servers in a data center. Hence, the query and data processing must be separable so that the latter can be offloaded to those servers and kept away from the client. Fortunately, separation is already ensured by the RV monitor synthesis. However, as mentioned in Section II traditional RV approaches feed the events through a finite state machine (FSM), which means that a framework would have to download and process every single event in a linear fashion. One approach that does this is called LARVA [14]. It could be argued that this does not use the full potential of a database management system (DBMS). Therefore, a goal of ARV is to synthesize the monitor that can be efficiently executed by a DBMS, i.e., using composed queries that returns only data that is relevant for the verdict and ideally can be parallelized.

V. Architectural Runtime Verification Concept

The fundamental idea of ARV is to utilize the runtime-verification-monitor paradigm for queries on architecture elements. Consider hypothesis H3 from Section III, which states that whenever the value of port 'io_wls' (see the lower left of Fig. 1) is set to enabled (i.e., the hazard warning light switch has been pressed), the 'io_bulb' ports of both actuators change to enabled within 100ms. ARV lets a user formulate this query, synthesizes a RV monitor and applies it to the runtime data. For this to work, architectural elements have to be associated with 'their' runtime data. Section IV-A requires that ARV supports different architectural models with different runtime information. ARV accomplishes this through a generic domain-independent model, which can represent structure and runtime of any domain-specific component-based architecture. This way, the same monitor can be applied to all data sets. It requires a *Domain Adapter* that transforms the domain-specific structure and runtime data into ARV's domain-independent model. The translated data is stored in a database. This implies that the monitor must be domain-agnostic as well. However, a user might still want to formulate hypotheses in a domain-specific way. Therefore, the properties must also be translated by the *Domain Adapter*. Fig. 2 shows an overview of the interaction between data acquisition (*Logging*) in the lower and *Monitoring* in the upper part.

The *Domain Adapter* must be supplied by a domain expert who knows both, the domain-specific architectural model and ARV's domain-independent model. To support a new architecture description it is sufficient to create a suitable domain adapter. Afterwards, the heterogeneous system that consists of different kinds of architectures and runtime-data can be transparently processed by the monitor. Users that are not familiar with the domain specifics of one part of the system can still use the domain-independent model to query it. In the context of RV of component-based systems, this is a novel idea that none of the investigated approaches [12]–[23] has considered so far.

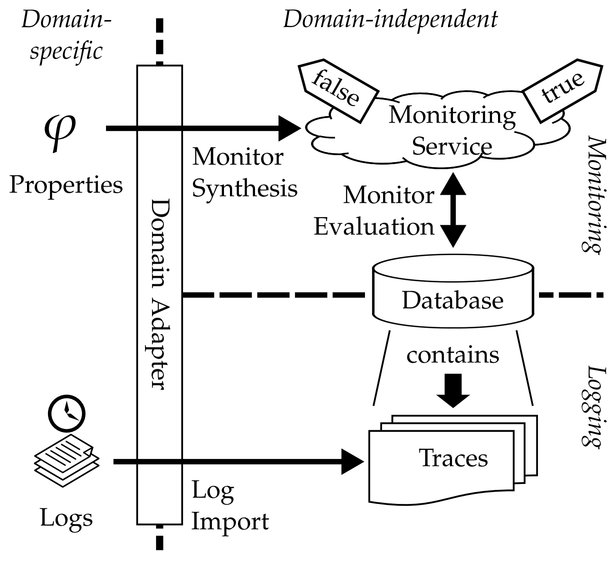

Fig. 2. Overview of the main approach

The next sections provide some details regarding the concrete model and how the RV monitor is synthesized.

A. Representing Structure

The domain-independent model that ARV uses to represent structure can be seen in Fig. 3. It has been designed to resemble what can be considered a common ground for any component-based architecture in a simulation-based test environment. We could not find an existing model that was better suited and comparably simple. However, it should be regarded as a first step.

The 'root' element is the *Configuration* that forms the container of one instantiated system. It may be composed of one or more components. A *Component* may contain other components (children), which makes them composite components. A *Port* is an interaction point between components. Components (and their ports) are actually instances of component types. This is in line with many component-based architectures and allows to reuse the component types. Also, verifying propositions regarding a component type has a much wider scope than regarding a specific instance of that type.

If the architecture in question has other element types that are relevant for runtime analysis, they will have to be mapped on those three just described. However, each of ARV's domain-independent model elements retains references to its domain-specific counterpart, so that no information gets lost. This is required for two reasons: (1) to translate domain-specific properties into domain-independent ones that can be processed by the monitor

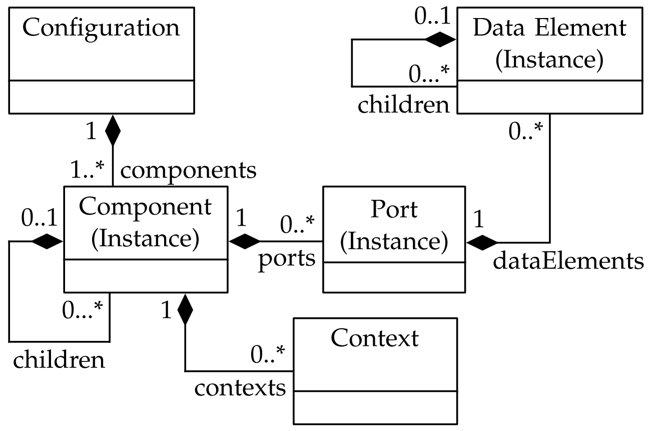

Fig. 3. The structural model

and (2) to translate the domain-independent results back into domain-specific ones for the user to inspect.

Finally, ARV adds another class named *Context* to this model. It cannot be found in common architectures. Its sole purpose is to form a bridge between structure and runtime. The name relates to the programming domain, where every function or statement is executed in a designated context (e.g., scope, stack frame, thread, task). Similarly, every component can have several (execution) contexts. A prominent example is, again, AUTOSAR, where the component's code may be executed in the context of different AUTOSAR Runnable Entities, which in turn are executed by different tasks. Using a designated class, there is no need to add runtime information directly to components or ports keeping the domain-independent structural model clean and separated from the runtime aspects.

B. Representing Runtime Behavior

A monitor in RV processes runtime events, i.e., every new event induces a new runtime state, which a monitor checks for certain conditions. ARV is no different in that regard, with the exception that it solely relies on a history of recorded events (*Trace*). Every event relates to a structural element of the architecture. This relation is realized through the *Context*, which has been introduced in the previous section. There are two event classes: (1) events that relate to a simple signal transmission from one port to another, which result in a value change at a port. (2) events that relate to the execution of the component's code. Here, the *Context* holds the relevant information about the execution state of a component. ARV uses the state machine depicted in Fig. 4 to represent those states.

In its simplest form, a component (i.e., its *Context*) can either be *running* or *suspended*, meaning its code gets executed or not. Thus, a component is expected to start in the *suspended* state provided that the recorded runtime data starts at the beginning. A change of the execution state relates to either a *start* or *terminate* event. Conceptually, the *start* event occurs when the code is called and the *terminate* event when it returns. Of course, such events and states can only be tracked if the architecture supports it and if there is some kind of instrumentation that generates the required logs (see bottom half of Fig. 2). However, ARV will still be usable if this information is not available.

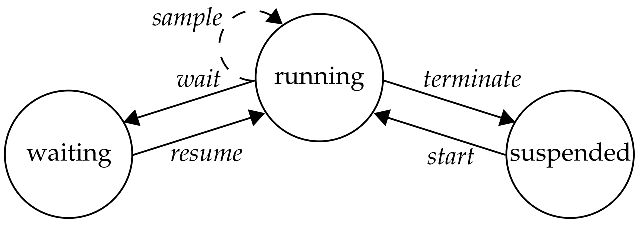

Fig. 4. The state machine of *Context*

For complex architectures like AUTOSAR that include a multitasking capable operating system, two states are insufficient. In real-time operating systems like OSEK, a task has a more complex state machine. Basic tasks have a further state *ready* and extended tasks have an additional state *waiting*. More details regarding task scheduling of real-time systems can be found in the OSEK/VDX standard [24]. ARV must consider the *waiting* state and its transitions *wait* and *resume*, because it can be a good indicator of resource deadlocks. The *ready* state on the other hand, is not considered by ARV. The *ready* state means that a task is prepared to run by the OS, but cannot, e.g., due to a task with a higher priority. On architecture level, i.e., from the perspective of the component, this state is not visible, because it is not a state of its execution context, but a state internal to the OS. This is also why on architecture level, this state is negligible. For this reason, ARV uses a state machine that contains only the states *running*, *suspended* and *waiting* as can be seen in Fig. 4.

ARV's runtime model (see Fig. 5) supports contexts that are executed in parent contexts forming a (*ContextStack*). This reproduces causal relations between the execution of components.

C. Verifying Properties

Following the requirements from Section IV-C, ARV distinguishes two types of atomic propositions. First, state propositions can refer either to one of the previously defined states of a component's context (see Fig. 5) or to the current value of a data element at a port. Values can be compared to other values for (in-)equality. In a formal language, this can be expressed using standard operators $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$. The second type of atomic proposition is the event proposition. It incorporates information about the current event in a specific context, i.e., start, terminate, wait, resume, or sample.

To specify temporal and logical relations between those atomic propositions (cf. Section IV-C), they must be combined using, e.g., a property specification language. There are different languages available that each has

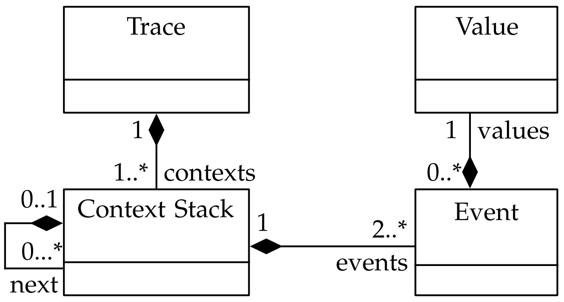

Fig. 5. The behavioral model

their own benefits and drawbacks. However, many RV approaches use LTL to specify properties. Furthermore, LTL has the advantage that several timed extensions exist that take real-time properties into account. Because of this and its familiarity, we have chosen LTL as base for the specification language in ARV. Thus, it supports the standard LTL operators **finally**, **until**, and **globally**. We added the bounded interval syntax and semantics found in the Metric Interval Temporal Logic (MITL) [25]. This allows a user to specify real-time constraints. Furthermore, the language supports the common logical operators **not**, **and**, **or** and **implies**.

D. Monitor synthesis

RV requires formal properties as input [5], [7]. A property φ is first automatically translated into a monitor (*monitor synthesis*) that afterwards observes the system's execution (*monitoring*). More specifically, it consumes events produced by the system and outputs a verdict that reflects the satisfaction of the property $Sat(\varphi)$. In contrast to other approaches, where the monitoring is performed at runtime, ARV applies it to traces that were gathered in preceding simulation-based tests. These traces are stored in a database that implements the model presented in Section V-A and Section V-B.

ARV performs a bottom-up monitor synthesis, where the formal properties are processed by a visitor that traverses the parse tree. This is similar to the approach by Maler et Nickovic [26], except that ARV processes discrete events instead of continuous signals. The result is a database query that returns the events in $Sat(\varphi)$. Compared to approaches like LARVA [14] ARV synthesizes complex queries that return only those events that are relevant, i.e., satisfy φ or none if no such events exist. This fulfills the requirement of Section IV-D.

Mapping the language operators presented in the previous section to the corresponding database query is straight forward for the logical operators, but requires some effort for the temporal operators. The former can be realized using set operators such as union or intersect. For the latter, we briefly explain the general approach exemplarily for the **finally** operator, which is just a special case of the **until** operator, but easier to illustrate.

Let $Sat(\Diamond_{[x,y]}\varphi)$ = $Sat($**finally** [x, y] $\varphi)$ describe the set of events that satisfy the property φ at least once within the interval [x, y]. To find this set, we have to go back in time from where φ holds. Consider our example hypothesis H3 (Section III). It could be formulated as $\Box(T \implies \Diamond_{[0,200]}\Upsilon)$, where T is the activation of the warn light switch and Υ the activation of both indicators. In words: It is always (**globally**) the case that an activation of the warn light switch **implies** that both indicators are **finally** active within 200ms. To find the relevant event set of the finally part we have to first get the events where both indicators are in fact active. Then, the earliest events satisfying $\Diamond_{[0,200]}\Upsilon$ would be those 200ms before. The latest events would be those right before the indicators are deactivated again.

In general, φ is not a single event, but can span multiple regions $[a_i, b_i]$. The set of events satisfying $\Diamond_{[x,y]}$ are therefore found in $[a_i$−y, $b_i-x]$ as illustrated in Fig. 6.

A naive approach is to query for each event in $Sat(\varphi)$ with timestamp t_j the corresponding events within the interval $[t_j$−y, t_j−x]. The union of all these sets then results in $Sat(\Diamond_{[x,y]}\varphi)$. However, querying all of these overlapping intervals requires many database operations yielding a bad performance. Therefore, ARV treats subsequent events as a union and only considers its boundaries (cf. timestamps a_1, b_1 and a_2, b_2 in Fig. 6).

VI. Implementation and Evaluation

We have evaluated the implementability of our approach based on the example presented in Section III. The first step to instantiate ARV was to create the necessary AUTOSAR-*Domain Adapter* as presented in the previous section. It consists of three parts: (1) a configuration adapter to insert concrete system configurations, (2) a log adapter to insert corresponding traces, and (3) a property adapter that allows the systems integrator to specify AUTOSAR-specific properties. Unfortunately, a detailed discussion of this cannot be given here, as this would require an in-depth discussion of the AUTOSAR standard. However, the inclined reader may examine the mapping of AUTOSAR elements to the domain-independent structural model in Table I and the mapping of AUTOSAR-specific runtime data (trace events) to the domain-independent events in Table II.

We obtained the runtime data using a dedicated logging mechanism that was added to the simulation platform[3]. It utilizes a feature of the AUTOSAR Runtime Environment (RTE) that allows to hook in callbacks. By applying the respective adapters, the AUTOSAR-specific runtime data has been translated into its domain-independent form and stored into a database.

The final step of the evaluation was to verify the actual hypotheses. Listing 1 exemplarily shows how H3 can be formulated using ARV's timed LTL grammar. It can be seen that the current language has its shortcomings. The value change from 0 to 1 must be formulated using an

[3]VEOS: www.dspace.com/go/veos

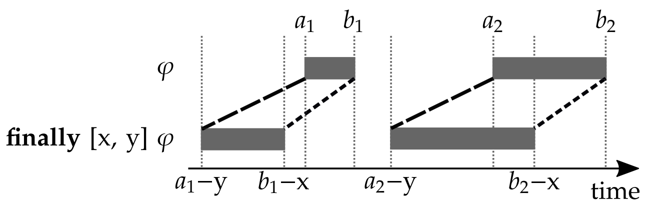

Fig. 6. Finally operator applied to a set of events.

TABLE I
Mapping of AUTOSAR elements to ARV's structural model

ARV Components	ARV port(s)	ARV context(s)
System	-	-
ECU	Com	Task(s)
SWC	RPort(s), PPort(s), PRPort(s), IRV	Runnable(s)

TABLE II
Mapping of AUTOSAR trace events to ARV's events

Category	AUTOSAR Trace Event	ARV Event
RTE API Trace Evs.	RTE API Start	set (only call)
	RTE API Return	set/get
COM Trace Events	Signal Transmission	set
	Signal Reception	get
	Signal Invalidation	sample
	Signal Grp. Invalid.	sample
	Com Callback	sample
OS Trace Events	Task Activate	sample
	Task Dispatch	start
	Task Termination	terminate
	Set OS Event	sample
	Wait OS Event	wait
	Received OS Event	resume
Runnable Trace Evts.	Runnable Invocation	start
	Runnable Termination	terminate

expression that involves finally with a one cycle interval. This is cumbersome and a dedicated operator would be preferable, which is considered future work.

```
globally (
 (DATA_ELEMENT@"WarnLightsSensor/out_wls/value" == 0 and
  finally[0, 10]
   DATA_ELEMENT@"WarnLightsSensor/out_wls/value" == 1)
 implies finally[0, 200] (
  DATA_ELEMENT@"FrontLeftActuator/bulb/value" == 1 and
  DATA_ELEMENT@"FrontRightActuator/bulb/value" == 1));
```

Listing 1. ARV timed LTL grammar example

We have used ANTLR[4] (ANother Tool for Language Recognition) to define the concrete grammar for the language and to generate a corresponding parser. For each node it creates the database query. Instead of generating SQL, ARV's implementation uses Microsoft's Entity Framework[5] (EF) to abstract the concrete DBMS. This allowed us to test the framework with MariaDB[6], Microsoft SQL Server[7] and the Azure SQL Database[8]. Furthermore, EF provides a LINQ[9] interface. LINQ allows one to compose queries and to defer the actual database access until a concrete event is requested (e.g., by querying the first or last element of a selected set). This way, the whole, or at least big parts of the parse tree can be composed into one single query. Also this is in line with the RV paradigm that a monitor can be synthesized once and then applied on multiple traces that have runtime data for the involved structural elements.

We implemented a graphical front-end that helps to configure the query and provides autocomplete for the property input. Furthermore, the user can select the desired domain adapter, so elements can be referred to in a familiar way. For example, in the domain of AUTOSAR it is common to reference elements by their path. Once a valid property has been entered, the RV monitor is synthesized. The user can then select a trace and have the verification performed. The results (if any) are presented as a table of runtime events that support the hypothesis, i.e., satisfy the property.

An in-depth performance analysis is pending. In this first step, it was not our main focus. However, there are some qualitative results. First, we found that the querying performance varied quite a bit between different DBMS, which was expected. Furthermore, we found that the complexity of the property and the implementation of the temporal operators have a big influence on performance. Some queries cannot be easily combined. In this case, the framework must download many events that form the base of the next query. This mostly happens when the proposition of a temporal expression yields many small sets of events that all need to be processed further. This not only degrades performance but also increases bandwidth requirements. Furthermore, we found that while Entity Framework is convenient w.r.t. the substitution of the underlying DBMS, it needs to be handled carefully to get a decent performance.

VII. Related work

We have investigated different RV approaches [12]–[23] w.r.t. their applicability to our requirements. It is remarkable that the need for RV at the level of software architecture has already been identified in 2001 by Mike Barnett and Wolfram Schulte at Microsoft Research [15]. They proposed a dedicated architecture description language named ASML. However, it only supports very specific client-server architectures and like many other approaches (e.g., MOP [18] or MOPBox [17]) has no built-in support for real-time properties. Approaches like RV-BIP by Falcone et al. [21] or the Runtime Reflection Framework (RRF) presented by Bauer et al. [12] inspired ARV. However, RV-BIP only works with BIP architectures and both require a dedicated monitoring component, which ARV does not.

Regarding the relational databases, LARVA [19] was our main inspiration. The authors argue that logging is performed in many systems anyhow and the idea to use SQL select statements helps to filter the events that need to be monitored. However, the original LARVA [14] has been designed for Java programs, not for architectures.

[4]ANTLR: www.antlr.org
[5]Entity Framework: www.asp.net/entity-framework
[6]MariaDB: mariadb.org/about
[7]Microsoft SQL Server: www.microsoft.com/sql-server
[8]Azure SQL Database: azure.microsoft.com/services/sql-database
[9]LINQ: https://msdn.microsoft.com/en-us/library/bb308959

VIII. Conclusion and Future Work

This paper contributes ARV—a new way of using RV on architecture level. It consists of a methodology and framework that enables a user to verify/refute hypotheses of system behavior using a language based on timed LTL. Hypotheses can be formulated 'beyond model boundary', because the underlying model is domain-independent. New architecture models can be integrated with the help of the *Domain Adapter* concept. Another advantage is that properties are translated into compendious database queries. Thus, in contrast to other RV approaches, where the client needs to process every individual runtime event, ARV allows to offload the processing to a standard database. We evaluated ARV using an AUTOSAR system. A graphical interface enables us to verify arbitrary hypotheses concerning the runtime behavior of its architectural elements. Finally, the framework supports multiple DBMS.

We consider the current state of ARV a starting point. In the future, many aspects have to be examined further. The evaluation is still at an early stage, especially regarding performance. It would be interesting to compare ARV's combined queries with the traditional approach to feed every single into a state machine w.r.t. execution times and the amount of downloaded data. Here, the parallelization of the queries can have an impact.

Furthermore, the underlying technologies (e.g., Entity Framework, relational databases) have to be further evaluated. It might be a better approach to use other database technologies and paradigms that are more suited to runtime data and its relation to the architecture (e.g., graph and time-series databases). Also, the domain-independent model can be optimized w.r.t. query performance. For example, verifying a hypothesis that involves several configurations or even traces with the current model requires many joins at the database level, which can be very expensive.

Another aspect is the query language. Even with the autocomplete feature, our LTL variant is still quite cumbersome to use in a productive environment. Other Languages, e.g., those used to query graphs might be more suitable to express typical hypotheses w.r.t. runtime analysis of integrated architectures.

Finally, we want to further evaluate the universality of ARV and its capabilities. Therefore, we want to apply the approach to an actual heterogeneous system in the aerospace domain.

References

[1] ISO, "Road vehicles – Functional safety," 2011.

[2] C. Baier, J.-P. Katoen, and K. G. Larsen, *Principles of model checking*. MIT press, 2008.

[3] A. Zeller, *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009.

[4] L. Stockmann, "Debugging models in the context of automotive software development," in *Proceedings of the Doctoral Symposium of the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems*, M. Chechik and D. Kolovos, Eds., 29 Sep. 2015.

[5] Y. Falcone, K. Havelund, and G. Reger, "A tutorial on runtime verification." *Engineering Dependable Software Systems*, vol. 34, pp. 141–175, 2013.

[6] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE, 1977, pp. 46–57.

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

[8] "System Desk Guide," *dSPACE GmbH*, 2016.

[9] F. Kirschke-Biller *et al.*, "AUTOSAR – a worldwide standard current developments, roll-out and outlook," in *5th VDI Congress Baden-Baden Spezial 2012*, Baden-Baden, 10 2011.

[10] T. Blochwitz *et al.*, "The functional mockup interface for tool independent exchange of simulation models," in *In Proceedings of the 8th International Modelica Conference*, 2011.

[11] SAE International, "AS5506 - Architecture Analysis & Design Language (AADL)," 2017.

[12] A. Bauer, M. Leucker, and C. Schallhart, "Model-based runtime analysis of distributed reactive systems," in *Software Engineering Conference, 2006. Australian*. IEEE, 2006, pp. 10–pp.

[13] S. Cotard, S. Faucou, J.-L. Béchennec, A. Queudet, and Y. Trinquet, "A data flow monitoring service based on runtime verification for autosar," in *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*. IEEE, 2012, pp. 1508–1515.

[14] C. Colombo, G. J. Pace, and G. Schneider, "Dynamic event-based runtime monitoring of real-time and contextual properties," in *International Workshop on Formal Methods for Industrial Critical Systems*. Springer, 2008, pp. 135–149.

[15] M. Barnett and W. Schulte, "Spying on components: A runtime verification technique," in *Workshop on Specification and Verification of Component-Based Systems*. Citeseer, 2001, pp. 7–13.

[16] H. Barringer, A. Groce, K. Havelund, and M. Smith, "Formal analysis of log files," *Journal of aerospace computing, information, and communication*, vol. 7, no. 11, pp. 365–390, 2010.

[17] E. Bodden, "Mopbox: A library approach to runtime verification," in *International Conference on Runtime Verification*. Springer, 2011, pp. 365–369.

[18] F. Chen and G. Roşu, "Towards monitoring-oriented programming: A paradigm combining specification and implementation," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 108–127, 2003.

[19] C. Colombo, G. J. Pace, and P. Abela, "Offline runtime verification with real-time properties: A case study," *Proceedings of WICT*, vol. 2009, 2009.

[20] M. d'Amorim and K. Havelund, "Event-based runtime verification of java programs," in *ACM SIGSOFT Software Engineering Notes*, vol. 30. ACM, 2005, pp. 1–7.

[21] Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem, "Runtime verification of component-based systems," in *International Conference on Software Engineering and Formal Methods*. Springer, 2011, pp. 204–220.

[22] L. Pike, A. Goodloe, R. Morisset, and S. Niller, "Copilot: a hard real-time runtime monitor," in *International Conference on Runtime Verification*. Springer, 2010, pp. 345–359.

[23] Y. Vandewoude, P. Rigole, D. Urting, and Y. Berbers, "Draco: An adaptive runtime environment for components," *Appendix of the EMPRESS deliverable for Run-time Evolution and Dynamic (Re) configuration of Components*, 2003.

[24] "Operating System Specification," *OSEK/VDX Version 2.2.3*, 2005.

[25] R. Alur, T. Feder, and T. A. Henzinger, "The benefits of relaxing punctuality," *J. ACM*, vol. 43, no. 1, pp. 116–146, Jan. 1996. [Online]. Available: http://doi.acm.org/10.1145/227595.227602

[26] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, 2004, pp. 152–166.