Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Die Klassifikation von Laufzeitverifikationswerkzeugen für Cyber-Physische und Adaptive Systeme hinsichtlich Reaktion, Interferenz und Anwendung - Eine Untersuchung

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Eugen Wagner
geboren am: 3.10.1997
geboren in: Minusinsk

Gutachter/innen: Prof. Dr. Lars Grunske
Dr. Thomas Vogel

eingereicht am: ..... verteidigt am: .....

Contents

1 Motivation

The idea of runtime verification (RV) in its most general sense revolves around extracting information from a running system with the goal of finding unexpected system behavior. There are several types of information and amounts of run-cycles for the system to consider when extracting information during runtime [32].

In this work, we focus on checking whether a running system satisfies a certain specification that has been set beforehand or not. A run-cycle is the traversal of a set initial system state all the way to a defined end state. Since RV tools are implemented differently, depending on how many run-cycles are considered, we do not set a specific limit or define a run-cycle in more detail. The goal is to analyze RV techniques using relevant tools and systems. The reason for setting this goal in particular is that surveys are scarce in the research field of RV and therefore only a few attempts at creating an overview of the most used or most relevant tools and approaches have been made. While attempts at taxonomizing and creating overviews have been made [32], to our knowledge, there have been no recent surveys about RV tools with a focus on recently developed RV tools and approaches.

1.1 Cyber-Physical Systems as a Domain

In this work, we attempt to create an overview of tools with applicability to real-world systems in mind. A suitable first domain to pick for RV experiments is the domain of cyber-physical systems (CPS), because a major challenge one has to tackle when thinking about CPS is real-time execution. Schwenger et al. describe three challenges one has to think about when developing RV approaches for CPS [70]. First, there is a continuous and infinite stream of events. Second, the local CPS clock may desynchronize from other clocks. Third, the physical components follow physical laws and therefore need to be considered.

There are countless challenges that methods in software engineering have to tackle in CPS. RV tools can be developed around a system's architecture in order to address some of these challenges. The first great challenge that comes to mind when thinking about CPS is the fact that there are many parts involved and there is often a complex architecture at hand. Each part of the system may have different specifications with different requirements that each need to be accounted for when attempting to apply runtime verification [77, 61]. Furthermore, timely synchronization in the regarded system poses a grand challenge as the tool has to account for it. Different system threads might be synchronized, or they might run asynchronously.

1.2 Adaptive Systems as a domain

Since RV promises to deliver important information about the observed system, there is an opportunity to take that information into account in adaptive systems. In the most simple form of an idea, information about errors within parts of the system could be utilized to self-adapt the system with the goal of minimizing the discovered error.

Furthermore, adaptive systems have become increasingly relevant as a domain for high-assurance systems.

Goldsby et al. see adaptive systems as a collection of steady-state programs and adaptations between them [38]. They clarify that steady-state programs are set up to fulfill environmental conditions without having an adaptive component. An adaptation is the transition between steady-state programs. There are a set of unique challenges that need to be considered in order to properly implement a runtime verification component into an adaptive system. Goldsby et al. utilize the special specification language called *adapt operator-extended linear temporal logic* (A-LTL) that has been specifically designed with adaptive systems in mind [38].

1.3 State of the Art

The research field of runtime verification has gained significant interest in 2010. While the International Conference on Runtime Verification has been active since 2006, research in runtime verification has only increased in the coming years. This makes the entire research field young compared to many other research fields in software engineering. Due to this fact, to our knowledge, runtime verification is still primarily focused on the theoretical aspects and less so on a real-world, industrial use. Software verification (SV) is a closely related topic to RV since specifications also play a fundamental role. The key difference between RV and SV, however, revolves around the runtime aspect. SV is often performed in theory to ensure the system's correctness. This work specifically focuses on RV tools.

In general, there are two major groups one can differentiate RV research into. The first group of research revolves around formal theory. As of today, there is still no standardized taxonomy for the field of RV. This means that there is increased ambiguity in the terminology of the field. Falcone et al. attempt to take a step towards fixing this issue by creating an overview of all the relevant RV areas, such as application, specification, and more [32]. With the help of a general glossary of terms in RV, researchers can more clearly describe complex formal structures and describe performed experiments on them. Kushwaha et al. for example, experiment with a framework in order to extend the trace of da monitor with the goal of reducing execution time and memory overhead [56]. However, here lies the research gap. While the first group of research attempts to improve a certain aspect of this taxonomized field of RV, there are few papers that contextualize and create overviews about the progress of RV and its applicability in real-world systems.

While the first group of approaches focuses on the theoretical foundation of RV, the second group of foundations aims to implement their findings directly into a tool. Furthermore, the second group also researches tools in an attempt to improve some aspect of them (e.g. usability, deployment, overhead). While this culmination of research is striving more towards the practical use of RV in a real-world scenario, there are still not enough surveys to give an idea about which tool is practical in what situation.

Bartocci et al. deployed the tool MoonLight in MatLab, together with datasets of

increasingly-sized data structures containing nodes and paths [8]. MoonLight is a great example of what is applicable for our survey, as it utilizes temporal logic equations to verify the configuration of a CPS. The temporal logic formulas were then set up in order to monitor the spatio-temporal properties of the data sets and simulate a real-world system. The issue with such work is the fact that often a tool is developed and compared against another tool or a data set. Such comparisons are helpful to test and evaluate the regarded tool; however they leave a research gap for a generalized overview about recent developments in RV tools.

1.4 Goal of this work

On the one hand, the goal of this work is to set up a general overview of recently developed or improved RV tools with the CPS and Adaptive Systems domains in mind. On the other hand, the goal is to compare the discovered RV tools under the aspects of interference, reaction, and application area. We note that while the two listed domains are taken as references for a real-world application of the tool, we are not evaluating these domains qualitatively for the application of the classified tools.

While a systematic literature review (SLR) was initially planned to reach the goals, the time constraint for this work does not allow for certain steps to be taken, like, e.g., the quality assessment. For a proper quality assessment, one is required to reach out to the researchers of the relevant scientific papers directly with the goal of ensuring the quality of their work. This is usually done in the form of an extensive questionnaire. Kitchenham et al. cover the steps that need to be taken in order to conduct a proper systematic literature review [52]. The clear disadvantage of a systematic literature review is the time and effort required compared to other types of literature reviews.

Due to this fact, we have decided to conduct a mapping study instead. The key difference in this type of study is the focus on structuring existing literature without evaluating its quality. It offers a review of studies with the goal of showing what is available about a certain topic, which fits our goals well. In order to conduct the mapping study, we followed the guidelines set by Petersen et al. [62].

1.5 Research questions

The goal of this work is to create a mapping study overview with a collection of recent and relevant RV tools. This will be done with the applicability of the tools on the CPS and the adaptive system domain in mind. The overview, together with the classification, will be the key contribution of this work. In the following, we propose the research questions that will be answered with this work.

- **RQ1 -** What is the desired and achieved goal of the tools?
- **RQ2 -** Which specification formalism is most commonly used in the RV tools?
- **RQ3 -** How do the tools handle discovered errors?

- **RQ4 -** How are the tools deployed?
- **RQ5 -** How are the tools evaluating the trace?

Further explanations about terms and definitions in the aforementioned research questions will be provided in Section 2. Furthermore, the data points required in order to answer the research questions will be discussed in future sections. With RQ1, we want to see in which direction the applicability of the tools is heading. With RQ2, we want to inquire about

1.6 Work Structure

The paper is structured as follows: In Section 2, we describe all terms and definitions relevant in order to understand the mapping study. Furthermore, we explain the different taxonomies for RV and why we decided to choose the three fields of applicability, interference, and reaction. Section 3 describes the process of conducting our mapping study, step by step. The data points will be explained in detail to function as a legend for the resulting literature table. Section 4 discusses the discoveries from the mapping study and how these discoveries relate to the proposed research questions. In Section 5 we conclude our work and discuss future work.

2 Background and Definitions

In the following section, we discuss and explain the necessary definitions and taxonomies of runtime verification.

2.1 Definitions

In order to reduce ambiguity among the terms of runtime verification, we explain our understanding of the definitions for the required terminology. Since countless studies utilize a slightly altered version of each definition that we use to discuss our topic, we deem it necessary to clarify our understanding of it [11].

Let us start off by explaining what we mean by a *system*. A system, in the context of runtime verification, is our domain. In our example, this can either be a combination of software and hardware in the case of a CPS [70] or primarily software in the case of an adaptive system [80]. A unit that we can regard and analyze within the system is called an *observation*. An observation is a step within the system's execution where the scope needs to be defined beforehand. For example, this could entail a single line of code as a breakpoint or the execution of a function within a program.

A *system trace* is a sequence of observations. There are multiple ways to set properties for the system trace. The most significant design decision one has to make in regards to the trace is whether it should be a continuous (infinite) trace or a finite trace under a certain time constraint. Both choices have ups and downs and may influence future design decisions. Further decisions revolve around what types of data are supposed to be extracted, ranging from signals to events or states.

Before the system execution even begins, one has to set certain *specifications* beforehand. Specifications are properties or goals that the running system has to achieve or uphold during runtime. These properties can range from subsets of the system trace up to a list of configured system properties. Every runtime verification method contains a *monitor*. In the most popular use case, a monitor is a program that analyzes the system trace of the monitored system and checks against the specification. One can imagine a decision-making process that goes through a list of non-complex specifications and gives information about the system's behavior [61].

The *overhead* in regards to runtime verification is the hardware load that the runtime verification method applies to the system during runtime. Examples of typical overhead in a tool could be the utilization of memory capacity within the system or the halt of a system thread during execution. This may lead to a decrease in performance regarding time or space.

The term *runtime verification framework* unifies several of the aforementioned terms. The RV framework consists of specifications set for the system and one or more monitors from the RV components [32]. It is the theoretical foundation on which any RV tool is built. Furthermore, a *runtime verification tool* is defined by an instance of a RV framework. We can conclude that any RV tool needs specifications as input to function. Optionally, there is also an interface or graphic generation to visualize the output of the tool, next to the monitor.

2.2 Taxonomy of Runtime Verification

The field of runtime verification is complex to explain as it contains numerous subcategories. Compared to other topics and fields in software engineering, RV is fairly young in relevance. Outside of the official International Conference on Runtime Verification by Springer, RV has not been in the direct focus of conference proceedings since around 2010. A substantially important first step for research in a certain kind of field is a unified, common taxonomy of the relevant terms in the field. When different researchers have a common understanding of the definition behind the name of a term, the associated scientific papers become a lot more accessible to the reader.

As such, Falcone et al. [32] attempt to lay down such a taxonomy of relevant terms in the field of runtime verification. In their research, seven major categories of RV can be filtered out. The categories *specification, trace, interference, reaction, application, monitor, deployment* represent the termonology in the field. Since attempting to classify tools against each individual category would prove far too extensive for this work, we have chosen the research questions with regards to interference, reaction, and application only (Figure 1). The decision was made with the application of RV to our domains, CPS, and adaptive systems in mind. First, application is universally relevant across both domains. Second, reactivity is especially relevant in adaptive systems since upholding the correct system state may require an enforcement of its processing. Third, interference is especially interesting in the CPS domain, since depending on how invasive the regarded RV approach is, it may enable new avenues to apply RV to the system.

Despite our evaluation not containing all seven of the categories, we discuss what each of them entails to help understand the extent of the field. This is necessary since each RV taxonomy is closely entangled with one another, and solely explaining application, reaction, and interference could lead to confusion in our experimentation.

2.3 Specification

The specification is the main input in a RV tool and, therefore, generally has to represent the desired system behavior as accurately as possible. The most important property of any specification is that it needs to be observable within the system (since it would be unfulfillable without this property). Any specification has a scope. Therefore, it can be applied to the entire system or subsystems within the entire system. Generally, it can be said that, on one hand, *decentralised* specifications can be more commonly used as input for RV tools since they are independent from the system's architecture [32]. On the other hand, *centralised* specifications need to be developed around the system's architecture. This leads to cases where RV tools have to be developed for the specific system at the expense of flexibility. Another commonly used input for RV tools is a stream of either events or signals, whose dimensions and scope within the system are dependent on the aforementioned factors.

One formal method to describe an informal specification and make it applicable for a tool's usage is to depict a property in *temporal logic* (TL) [19]. Temporal logic has

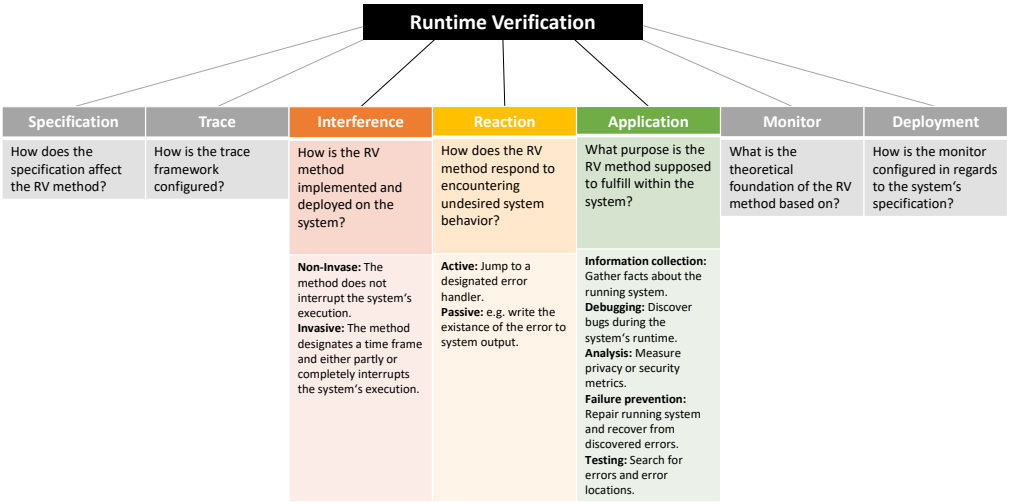

Figure 1: The seven categories of runtime verification, taxonomized by Falcone et al. [32]. The focus of this work, however, will be on application, reaction, and interference. Note that this overview does not consider intersections between different subcategories.

the upside of being able to create automata out of a TL formula and vice versa. The most commonly used form of TL is *linear temporal logic* (LTL), which is derived from first-order logic and finds countless uses in model checking. This is due to the fact that LTL formulas can contain information about how a path is traversed in a graph model.

Tools such as Java PathFinder are built on the foundation of LTL, on one hand [47]. On the other hand, it allows a specification input from future and past time temporal logic. This was achieved by implementing the use of future time operators on the already existing propositional operators that have been derived from first-order logic. Since we will not be analyzing the specification aspect of the regarded tools, we will not be going into further detail about the different properties of temporal logic and other forms of specification input.

2.4 Trace

The trace part of the taxonomy refers to a subset of the system trace, which entails a subset of the sequence of observations within the system. This, of course, only refers to the case in which the trace is finite, containing a start and an end point in the sequence of observations. The abstracted form of a trace is a *trace model* [32]. The trace model is a separate program that allows the configuration of properties regarding time constraints and what types of data are to be compared against the specification. This entails that the trace model generates our desired subsets of system trace.

Once the desired observations in the trace are extracted, they can then be used as input for the RV monitor. Falcone et al. refer to this case as *sampling*. Sampling can occur based on time or on specifically set events [32]. The sample is then a subset of the observations in the model. Further statements and information about a single sample revolve around how many distinct system behaviors are covered by it. It is then deemed *precise*, if it covers all types of system behavior, or *imprecise* in the other case.

Reger et al. explain that any specification ϕ has a set of possible traces assigned to it [66]. Since traces are so closely tied to specifications, they naturally need to match the specification language.

Example 1 (Propositional Traces) Given a specification from a finite-state automaton in the form of a regular expression, it becomes clear which trace corresponds to the specification and which doesn't (see Figure 2).

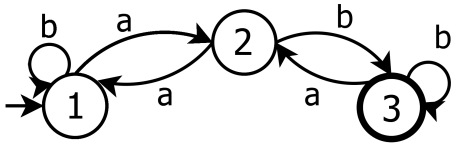

Figure 2: A deterministic finite-state automaton to visualize how a specification may be modelled in order to check trace samples against.

In this example of a deterministic finite-state automaton, there are three states: 1, 2, and 3, with the two terminals a and b. The language model with the accepting words of this automaton is one possible specification ϕ. The regular expression describing the accepting language of ϕ is $(b|aa) * a(aa|ba) * b(ab|b)*$. It's clear to see that the trace bbab corresponds to ϕ, where bbaba does not [66]. In real-world use cases for CPS or adaptive systems, however, regular expressions do not find much use since they fail to integrate time components into the specification language.

For this reason, the usage of temporal logic formulas is far more common in the development of a tool for the extraction of trace or setting a specification. As mentioned in Section2.2.1, the most common form of TL used for RV tools is LTL.

LTL is based on first-order logic and therefore contains the following.

- variables $a, b, c...$
- logical operators $\neg, \wedge, \vee, ...$
- existential quantors $\exists, \forall$
- temporal operators $\Box, \Diamond, ...$

The temporal operators correspond to an action that needs to be taken in order to fulfill the formula. In LTL, however, this does not guarantee the desired time component in the specification.

Example 2 (LTL for event ordering) The specification $\Box((\neg a \vee \Diamond b) \rightarrow c)$ is written in

the LTL specification language and has been inspired by the example from Reger et al [66]. The *global* operator $\Box$ guarantees that any trace has to hold at some point, and at any followup state on the following path, this state stays the same. Possible traces for this specification are the sets $\{\}, \{\neg a\}, \{b\}, \{\neg a, c\}, \{b, c\}, \{c\}$. The *finally* operator $\Diamond$ guarantees that there is a certain state in the future where the trace finally holds but may change states again after the hold. This example shows that LTL adds an ordering component to logical formulas but not a time component directly. For this reason, there are further detailed derivations of LTL, like, for example, *Metric Temporal Logic* (MTL) [30]. In MTL, there is the added component of intervals to ensure the timed availability of the subformula in the TL formula.

2.5 Interference

Falcone et al. come to the conclusion that there are two subcategories in the interference category [32]. In the context of our work, runtime verification tools can then be either *invasive* or *non-invasive* in regards to the regarded system domain.

Note that interference is measured in a more qualitative way, and therefore, a RV tool can't be classified as invasive or non-invasive with total confidence. The two categories should be viewed as a spectrum instead. A more invasive tool could increase the system's overhead during runtime. For the CPS domain, the measuring of CPU temperatures can be deemed invasive. Naturally, it is dependent on the system's architecture to measure the degree of invasiveness, since sharing the same memory space between the RV tool and the system may also be deemed invasive. As with the other taxonomies, interference is deeply entangled with other categories like specification, trace, and monitoring.

For example, the way the monitor is implemented in the system can affect the amount of system overhead [76]. The way trace samples are extracted from the system can also interfere with the system's runtime by being directly integrated into the code. In the context of this work, interference will be understood as the manipulation or accessing of physical components within the system, like the CPU or RAM. Consecutively, this means that interference will be less relevant for the adaptive system domain as physical components are less likely to play an integral role within the system.

Implementing runtime verification with a low amount of interference is a central research topic for future research. The very concept of runtime verification is designed around reading and accessing threads of the regarded system, and therefore, interference is impossible to avoid entirely.

Berkovich et al. propose several concepts in the form of two algorithms with the goal of reducing the runtime overhead of the regarded system [14]. They efficiently separate monitoring assignments from the running program with parallelized hardware processing units, which directly reduces interference. During the experimentation, they let the system (which in their case was a program) run on a CPU and the RV monitor run on a GPU. The monitor operated with worker threads that analyzed the program trace and monitored multiple properties simultaneously.

2.6 Reaction

Reaction, in the context of runtime verification, describes how the monitor handles behavior in the trace that differs from the specification in the regarded system. Like interference, the field of reaction is closely tied to other fields in RV. The reaction taxonomy is directly tied to the monitor as it reflects the implemented behavior of the monitor. Furthermore, interference plays a role in how the monitor is allowed to halt system threads or components in order to react.

In the most common sense, a reaction from a RV monitor can either be *passive* or *active* [32]. Should undesired behavior within a trace be identified by the monitor, in the case of a passive reaction, the monitor just gives information about the discovered inconsistencies through the output.

In the case of an active reaction, the monitor directly accesses the running system during runtime [44]. This, of course, implies that the monitor has to be running in accordance with the system and cannot be monitoring it offline. An active reaction could be implemented in different ways. From jumping to a specific point in the code to handle the error to altering the system's output altogether.

2.7 Application

The application area describes what purpose the RV tool is supposed to fulfill within the system. The first purpose that may come to mind is information collection. The main purpose of a RV monitor is to analyze traces from the regarded system and compare them against the specification to come to a conclusion. Based on the conclusions, information about the system can then be communicated to the user. Furthermore, this information can be visualized through a graphical output, which may help to make statistical statements about bottlenecks or the correctness of the system. Shafiei et al. have presented a RV approach in the MESA tool [73]. The tool is able to give descriptive output with graphical visualizations, making it applicable for testing and debugging tasks.

With the RV tool's architecture being so closely tied to the system's design, it is also possible to make performance tests or debug parts of the system [32]. This, naturally, also opens the possibility of analyzing system vulnerabilities in regards to security. Depending on the implementation of reaction behavior in the tool, the main purpose of the application can be to discover system failures in time and either notify the user of them or directly influence the system's processes.

We refer to Sánchez et al. for information about design choices developers have to make before attempting to implement a RV tool for CPS domains [77]. Efficiency plays a key role in RV as it can reduce time or space overhead. They discuss the problem of monitoring efficiency, as having one central monitor for all the computation is not the most efficient solution. Generating local monitors for specific sub-formula traces can prove far more efficient, but it comes at the expense of synchronization and accessibility challenges that need to be addressed.

2.8 Monitor

The monitor is a key component of RV. It takes trace samples from the system as input and compares the contained information within the trace against specifications within the system. As mentioned before, the monitor is a program with the goal of making decisions about the traces it analyzes.

Falcone et al. explain that a monitor can either be based on *automata*, *graphs* or *rewrite-based*. Automata-based and graph-based approaches entail that an algorithm checks the system trace based on finite-state automata or graphs and simulates the execution in order to check for violations [32]. Rewrite-based means that the very foundation of the monitor is based on grammar rules from the specification language [45].

Further design decisions revolve around how the monitor extracts and analyzes trace samples. For example, system specifications can be extracted, and from the specification language, an automaton can be generated [1]. With this automaton, it is clearly possible to check if a trace is running into errors or not by simply inputting the trace as an input word to the automaton. Based on the results of the simulation, a verdict on whether the execution was correct or faulty can then be made.

The opposing design choice would then revolve around storing tables of data that each represent one observation. Utilizing relational algebra, operations like union can then be used to check entries in these databases against a specification (default case behavior) inside the system. Relational databases can easily be evaluated this way by inputting observation entries into the specification.

Falcone et al. discuss the output design of a monitor through the terms *sound* or *complete* [32]. On one hand, a monitor with the completeness property guarantees an output. While this output does not guarantee correctness, it offers a justification for its work. On the other hand, a monitor with the soundness property guarantees a correct output.

There are more terms necessary to cover the monitor taxonomy part of RV. However, in our work, we do not plan to discuss the monitor design taxonomies of the regarded tools. Therefore, we do not go further into these definitions.

2.9 Deployment

The deployment part of the taxonomy directly correlates to how the RV monitor is deployed onto the system. This, of course, heavily influences the feasibility of RV approaches.

To summarize, deployment covers the terminology of how monitors are implemented and how traces are extracted. Furthermore, it covers timely components within the system, like, for example, synchronization between a system thread and a RV monitor.

In the case of a synchronized monitor, the RV approach is able to extract a trace sample from the system, analyze it, give out verdicts, and extract the following sample without "falling behind" during runtime [32]. Conversely, an asynchronous RV approach would force parts of the system to halt in order to have time for the trace analysis.

Once the analysis is complete, the system is notified by the monitor to resume the halted processes.

2.10 Related work

The field of RV primarily revolves around researching formal methods. To our knowledge, Falcone et al. have created the most extensive survey about RV tools, structuring the developement progression in the field from 2016 to 2018 [31]. They furthermore verify their findings with a questionnaire, regarding the classification, that most of the authors of the analyzed tools have filled out [32].

Bures et al. have conducted a mapping study regarding interoperability and integration testing methods on the Internet of Things (IoT) domain with a set of 102 papers [16]. While this is related in regards to analyzing RV approaches on a domain, their work differs from ours. First, we are assuming the application on CPS and adaptive systems. While the IoT domain has similarities, the characteristics differ in regards to the architecture between CPS and IoT [70, 16]. Second, we are extracting documentations and information about RV tools, not theoretical approaches.

Sánchez et al. have created a survey to showcase and overview the most state of the art in regards to monitoring techniques [77]. They analyzed areas of application (domains) to be used and collected relevant information in order to formulate significant research challenges for RV monitors. Our survey is focusing on an implementation approach past the design stage, assuming that a theoretical idea has already been implemented into a tool.

Mapping studies and surveys are generally few in the field of RV. In our research, we were unable to find recent research papers, regarding mapping studies or literature reviews of RV tools and their applicability in recent years. This leaves a structural research gap, as it is difficult to understand the current state of the art and progress on recent tool developments. This is especially true to researchers, that are newer to the field of RV.

3 Mapping Study

As previously mentioned in Section 1.4, in order to get results and make statements about our proposed research questions, we conduct a mapping study. We follow the guidelines proposed by Petersen et al. [62], since the goal of their work is to find best practices in conducting a mapping study from existing literature on the topic. The outcome of a mapping study is a map that displays the structure of the regarded literature.

The goal of the mapping study is to give structure to newly developed or updated RV tools and show their applicability in the CPS and adaptive system domains. We primarily focus on RV tools directly, which means we only include research papers that present a tool or extend an already existing one. We acknowledge that this kind of approach may exclude frequently used tools for our regarded domains that have been developed before our regarded publishing year timeframe (2018-2022).

The process followed in this section, bundled with the extracted artifacts, is visualized in Figure 3.

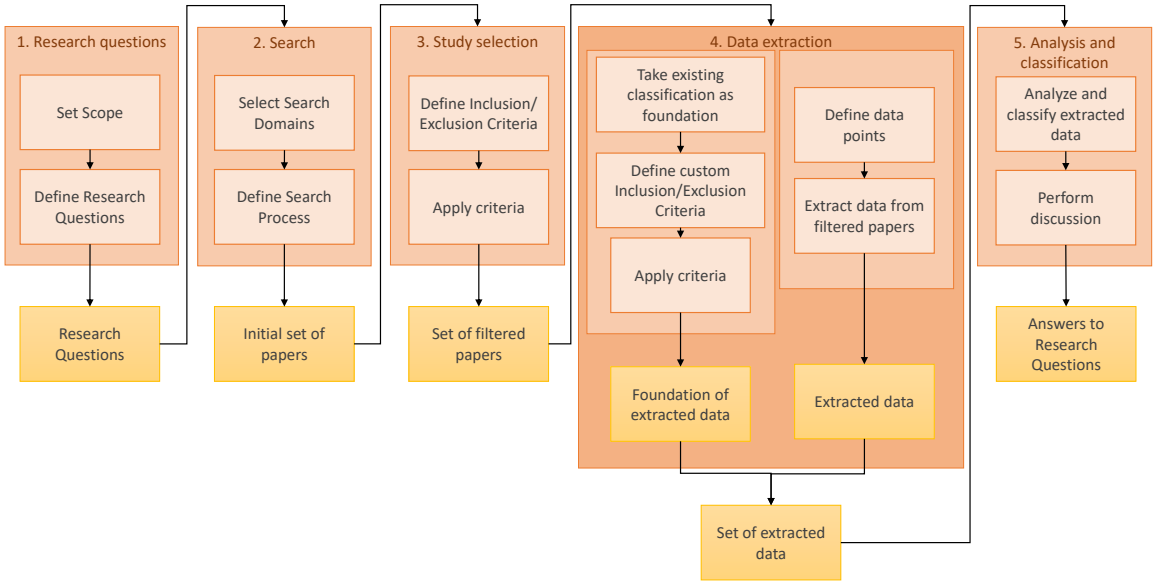

Figure 3: The process of the mapping study conducted in this work. We follow the steps of Petersen et al. in the definition of stages [62] and Bures et al. in the visualization of the stages [16].

3.1 Threats to Validity

Since this work is conducted by a single researcher, we note a potential threat to the theoretical validity of the mapping study. There is a potential for researcher bias as

there is no second person reviewing the mapping study process. Furthermore, there is a chance that relevant information may be overlooked, incorrectly understood, or not extracted to an adequate extent. Due to the nature of this work, it is not possible for us to utilize control actions in order to combat these threats.

Descriptive validity describes whether information from research has been observed objectively or not. Petersen et al. see qualitative studies as having a higher threat to descriptive validity than quantitative studies [62]. Quantitative studies make statements based on statistics and mathematical proof, whereas qualitative studies may make statements based on observations. To combat this threat, we structure our findings and observations in the form of an Excel sheet and a workspace in the markdown document management tool Obsidian.

Finally, we acknowledge the fact that we do not have access to the conference proceedings from the Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). While potentially relevant papers discussing RV tools may be contained, we do not have the means to regard them in this work.

3.2 Searching Process

In the first step of the mapping study, we describe how search strings are formed in order to select and find research papers. In the next step, the inclusion and exclusion criteria will be defined and later applied to filter out papers that may not be relevant to our study. And finally, the remaining set of papers will be used for the classification.

We continue the work of Falcone et al. [32] by adopting their findings as a baseline. They classified around 60 tools against one another in a similar way to the steps of a mapping study. The problem, however, is that the tools were only regarded in the time span of 2016-2018 (publishing years), leaving a research gap up to the current year. Their work is from 2021 because a questionnaire was additionally conducted on top of the classification results from an earlier work [31]. Survey research about RV tools is few and far between in the field, and to our knowledge, no further survey has been conducted in regards to runtime verification tools in recent years. Therefore, we apply customized exclusion and inclusion criteria to the 60 tools and include the remaining set of tools into our set of regarded tools and papers.

In order to limit and structure the search, we only consider the proceedings of three different international conferences. First, we search for papers in the yearly proceedings of Springer's *International Conference on Runtime Verification* from 2018 up to 2022. The conference proceedings contain a quantity of scientific papers ranging from 10 to 40. We deem this search domain to be the most important one for our research goals, as formal methods and the concepts of runtime verification are the most prevalent in the proceedings of this conference.

Another search domain is the *International Conference on Software Engineering and Formal Methods* from 2018 to 2022. This conference by Springer will also contain several relevant research papers, as the proceedings often contain an entire section about runtime verification or model checking.

The last search domain will be the *Symposium on Software Engineering for Adaptive*

and Self-Managing Systems (SEAMS) from 2018 to 2022. We estimate a far lower amount of RV tool papers that we could use for our research from this conference, but we still want to cover this search domain since we are interested in the perspective of adaptive systems. The SEAMS conference offers the challenge of accessibility. Since this work is conducted with the help of Humboldt-University partnerships, we do not have access to the majority of SEAMS proceedings as they are not partners of this university. Naturally, these papers will not be considered in our work.

The main challenge of searching through each conference's proceedings will be the proper selection of exclusion and inclusion criteria. On one hand, we want to avoid filtering out potentially relevant research papers. On the other hand, we want to accurately represent all the relevant RV tools in the regarded time span.

3.3 Keyword Identification

Since we will be going through the entirety of the proceedings in the three conferences, a search, as explained by Petersen et al. will not be conducted [62]. In their work, they propose defining a population, an intervention, a comparison, and the outcomes beforehand. Then they overview the different search strings in order to find relevant papers. This sort of approach makes sense when the search domain is a large database or search domain containing a high quantity of data entries or papers.

The selection of scientific papers in our work, however, will be solely based on the filtering of the following inclusion and exclusion criteria. The reason for this decision is that, with the resources available to us, it is possible to apply the inclusion and exclusion criteria to each paper in the conference proceedings within the time frame of this work. Therefore, a search is not required.

The inclusion and exclusion criteria are chosen so that only papers with a heavy focus on RV tools will be in the resulting set of papers.

3.4 Study Selection

The data filtering in regards to this work is applied at two different points. The first is the classification from Falcone et al. on the 60 RV tools presented in their work [32]. The second being the conference proceedings from the SEAMS, the SEFM, and the RV venue. Since the data on a classification survey and unfiltered conference proceedings are different by nature, we cannot simply apply the same inclusion and exclusion criteria to each of them.

In the following, we describe what each inclusion or exclusion criteria entails for the conference proceedings. The items in the following list start with a criteria code, containing an I (inclusion) or an E (exclusion) combined with a number. Following the criteria code is the title of the criterion, which is then followed by a description.

- **I1 - The research paper is written in German or English.**

  This inclusion criterion guarantees that the resulting set of research papers

only contains papers that we can understand without a language barrier. Since our search domains exclusively contain international conferences, however, all the initially regarded research is written in the English language.

- **I2 - The main topic of the research paper revolves around a tool.**

  The goal of I2 is to include initial relevant research for the classification. Papers that merely mention a tool do not fulfill the criterion, as we want to evaluate the current field of RV in regards to what tools are applicable to CPS or adaptive systems. Therefore, only research where tools are explicitly discussed, implemented, or extended is included under this criterion.

- **I3 - The focus of the research paper is set on a concept of RV.**

  In combination with I1 and I2, this criterion filters in the cases where we have a research paper about a tool with a concept of RV in mind. The venues for conference proceedings may have a tool paper with a topic in software engineering that does not directly correlate to RV. The focus of our work has been to determine the relevant RV tools from recent years. I3 guarantees that we collect the papers we need in order to properly conduct a survey about RV tools.

- **E1 - The research paper is not written in German or English.**

  We exclude any research paper written in a language other than German or English. Naturally, this is due to the language barrier in understanding research in other languages. While this exclusion criterion is not applicable to any of the research we've reviewed, we find it necessary to include it. In the case of conducting future work similar to ours, these criteria can be used as motivation to adjust the filtering of different venues.

- **E2 - The main topic of the research paper does not revolve around a tool.**

  This criterion excludes research that we cannot find use for in the classification of RV tools. By tool, we mean an RV approach that has been experimented on before and has been implemented into an accessible tool. This implies that it is beyond the design phase and already has a version that is applicable to a system or program. Furthermore, we are excluding research that improves on some concept of RV in a certain domain with experimentation, tailored specifically for the problem evaluation in the paper [18].

- **E3 - The focus of the research paper is not set on a concept of RV.**

  With E3, we are excluding papers that cover a topic in software engineering

without directly going into detail about RV. The reason for this is that there are tool papers about software verification that do not directly revolve around RV. While there are similarities between the former and the latter, the concepts are fundamentally different. Combined with E2, we are guaranteeing that we filter out papers without enough of a connection to a concept of RV and a practical approach in the form of a tool.

The aforementioned exclusion/inclusion criteria are applied to the conference proceedings of different venues.

In the following, we describe the specialized inclusion criteria for the taxonomy of Falcone et al. [32]. Any tool, not covered by the inclusion criteria, is automatically excluded. The aforementioned criteria do not apply to this filtering.

- **FI1 - The author's input has been given in the questionnaire of the tool.**

  Falcone et al. have created a questionnaire with around 30 questions, attempting to cover the taxonomized field of RV [32]. This inclusion criterion includes all tools where at least one tool's author has filled out the questionnaire. The goal is to include tools where the tool's authors have expressed their side of how the classification process is being conducted. This will improve the accuracy of the classification.

- **FI2 - The tool's fitness is above or equal to 80%.**

  Falcone et al. have conducted a questionnaire with 28 questions that has been sent to the creators of the tools they have classified [32]. This was done in order to make a quality assessment and get an opinion on the classification conducted in the paper. Each question had the option of being deemed "not applicable" (unfit). A tool's fitness percentage was then based on how many questions weren't unfit for the tool's authors. We attempt to set a threshold to find tool entries in the work of Falcone et al. where the tool's authors mostly agree with the classification. A low fitness score indicates that the author disagrees with parts of the questionnaire, which may affect the accuracy of the classification.

- **FI3 - The tool is publicly accessible in 2023.**

  Since the work from Falcone et al. has been conducted up to the year 2018, several URLs listed in the work no longer work [32]. Reasons for this vary, from content being moved to a different platform to tools not being maintained over recent years. We expect to filter in tools with working URLs, leading to either a git repository containing the source code of the tool or a download page containing relevant documentation. This is necessary, as we want to find as much information as possible for each tool. In the case of a privately contained documentation or website of the RV tool, we can only analyze the referenced research papers [72].

Any paper that meets all the inclusion criteria is accepted into the classification as a tool. If one or more exclusion criteria are met, the research is automatically rejected from further consideration.

3.5 Data Extraction

In the following section, we define what data items we are planning to extract from the papers. As a prerequisite from the previous sections, we already have a filtered set of RV tool papers to classify. In this step, we define what aspects and data the papers are classified by. Next to the papers classified by us, we are reviewing the set of filtered tools from Falcone et al. [33] again to check for changes in development.

The resulting table from the data extraction will yield the classification of the aforementioned tools. The table will allow us to evaluate the recently developed or extended tools based on application, interference, and reaction. Furthermore, the resulting evaluation will yield answers to our proposed research questions from Section 1.4.2.

In a similar way to Section 3.2.3 we list the distinct data points that we are aiming to extract from the research papers.

- **DP1 - Area of improvement within the system**

  With DP1, we want to see what application area the tool is fit for within the system, covering the application part of the taxonomy. This could range from primarily information collection to failure prevention. Further possibilities include the debugging or testing process within the system, where the tool takes on a supporting role [10]. Another possibility is property verification. To summarize, we're extracting data regarding the use case of the implementation. DP1 contributes answers to RQ1.

- **DP2 - Proposed objective of the tool**

  In DP2, we are checking for the proposed objective in the research work itself. Unlike DP1, we focus on the proposed research questions to compare the actual tool's behavior and applicability against the proposed goal of the tool's research. DP2, combined with DP1, answers RQ1 by providing a direct comparison between the actual implementation and functionality of the tool against the prior research question of the author or researcher. Possible values include scalability, accessibility, reactivity, overhead efficiency & performance, and lastly, correctness.
  Scalability entails the feasibility of extending the tool alongside the growing system architecture of the regarded system. With accessibility, we extract how simple the tool is to deploy, run, or utilize for less experienced users or engineers. This could, for example, include a GUI for direct interaction with the tool. The

value of reactivity describes the goal of making the tool efficient at detecting errors in a timely manner. The of value overhead efficiency or performance describes the goal of minimizing the overhead or reducing execution time during trace analysis (for example). The goal of correctness revolves around performing a formal mathematical proof of the algorithm the tool is based on in order to ensure correct functionality.

- **DP3 - Specification functionality**

  Under specification functionality, we look at the theoretical foundation the tool is built on. We are looking at what the specification formalism is by checking the specification language that is understood by the tool. Examples for this data point include finite-state automata with a model-checking-based approach to verify a specification. Further examples include TL formulas to verify a trace. Further variations of TL and automata-based approaches exist and will also be considered. There is no predetermined set of values fit for this data point, as countless tools use a custom specification language. DP3 will provide data in order to answer RQ2.

- **DP4 - Output**

  The data point DP4 will extract how a tool communicates its discoveries to either the user or the system. Possible examples of distinct output formats include terminal output, generated logfiles, and graphical output. Next to DP3, this data point will offer answers to RQ5. The specification formalism directly correlates to the feasibility of how output can be generated. In order to extract DP4, we use the same structure as Falcone et al. but include the classification of graphical output [32]. There are three different possibilites that the value in DP4 can take on. The first possibility includes a single verdict, witness, or robustness. The second is a sequence of verdicts, witnesses, or robustnesses. And lastly, a graphical output, including a plot or another visualization of the output data.
  A verdict in this context means a boolean constant that decides whether the observed trace matches the specification or not. A witness contains increased descriptiveness compared to the verdict as it directly offers the variables that violated the specification. Robustness output attempts to assess how much a specification has been violated or satisfied within the read trace.

- **DP5 - Interference**

  Under DP5, we are looking to extract information regarding the intrusiveness of the tool within the system. Examples for this data point include sharing the same memory space as a thread within the system or directly accessing a part within the system. In this context, DP5 includes accessing physical and software components. Naturally, this indicates an invasive tool. A non-invasive tool

could, for example, access the system's trace through an API designed to access logged data without interfering in its processes. With DP5, we are extracting information in order to answer RQ4.

- **DP6 - Active reaction**

  The data points DP6 and DP7 directly handle the reaction part of the tool. When undesired or unspecified behavior is discovered by the tool, how is the situation handled? Examples of an active reaction include an exception, where the tool jumps to an exception handling point in the program and intervenes in the system in some way. Further possibilities include backtracking (rollback), which involves reverting the running system to a previous state before the error occurred. Further possibilities include enforcement by the tool. Examples could include writing back and altering information streams within the system in order to prevent a failure. With DP6, we are looking to extract information in order to answer RQ3.

- **DP7 - Passive reaction**

  Similar to DP6, DP7 extracts information about what the tool does in the case of discovering unspecified behavior. A passive reaction makes sure that the system's execution isn't halted. This implies that the tool is restricted in its ability to contain errors. Examples of a passive reaction include an explanation, where the tool generates terminal output or a log file listing the discoveries for the user. In the most common case, a tool reacts passively by outputting a log file or writing to the terminal with the results of the specification analysis. This could be extended with additional information about the number of satisfied or violated specifications (statistics). With DP7, we are looking to extract information in order to answer RQ3.

- **DP8 - Stage**

  In DP8, we are looking to extract information about when the RV tool operates. There are two possible values for this data point: online and offline. In the most common case, when talking about a stage, a RV monitor is meant. An online monitor runs during the system's runtime. The opposite monitor, running offline, reads log files from the system and analyzes the trace while the system is not running. This data point is supposed to deliver answers for RQ4.

- **DP9 - Synchronisation**

  Similarly to DP8, DP9 directly correlates to RV monitors. A monitor can either run synchronously or asynchronously with regards to the system. A synchronous monitor can operate next to the system and does not desynchronize

in its execution time to avoid data races. Asynchronous includes monitors that are implemented with system halting in mind. For example, the tool extracts some fixed-length trace from the system and enforces a system pause in order to analyze the trace. Once the tool is finished with the analysis, the system gets notified to resume its execution. With the data from DP9, we can answer RQ4.

3.6 Analysis and Classification

In the final step, we attempt to visualize every value in each data point and structure our findings from the classification. The visualization yields answers to our proposed research questions (RQ1, RQ2, RQ3, RQ4, RQ5). Furthermore, we discuss our findings in two ways. On one hand, we discuss the results in regards to the basis of the tools we are re-evaluating [32]. On the other hand, we attempt to give an overview of tools that have been developed from 2018 up to 2022 in our search domains.

4 Classification of tools

This section contains the main contribution of this work, as it captures our experiments described in Section 3.

Since we have not defined any search strings, we instead commit to reviewing all proceedings from the SEFM and the RV in the time span from 2018 up to 2022.

4.1 Mapping study process

In total, we have taken 236 papers into consideration. From the RV conference, 123 papers were included. From the SEFM conference, 113 papers were included. While initially it was planned to include the SEAMS conference as well, we have seen only a handful of publicly accessible papers in their conference proceedings. Since we do not have direct access to the proceedings, we decided to exclude this venue as a search domain.

After applying the inclusion criteria (I1,I2,I3) and exclusion criteria (E1,E2,E3), a total of 67 papers were taken into consideration. Out of that set of papers, 46 papers were chosen from the RV conference and 21 papers were chosen from the SEFM conference (see Figure 4). Additionally to the data filtering criteria, we've extracted the paper title, publishing year, authors, venue, page numbers, and doi URLs. Furthermore, we extracted the name of the RV tool (only if one was presented). Some of the listed tools, like e.g., UPPAAL [57, 60, 53] or TeSSLa [50] are part of a larger framework or are frameworks directly. In such cases, we've only included them in the case of a RV tool being part of the framework's toolkit.

Additionally, we included information about the paper's "type". This is done for the sake of overviewing whether a RV tool is presented or extended within the work. The third possible value for type is "theoretical". Any paper that does not directly include a RV tool is automatically deemed theoretical since it is not relevant to our classification and therefore only potentially offers theoretical information about RV. Lastly, type can take on the value "tutorial", where the paper showcases a tool's usage extensively through a practical example. The data filtering and the data mapping artifacts are accessible through our git repository [1].

In the following step, we performed the data extraction. First, we regarded all 60 tools from Falcone et al. [32] and applied the customized filtering criteria (FI1, FI2, FI3). We designed the criteria with recent information in mind or existing information that has changed over the years. Tools that no longer have development support and are no longer available were supposed to be filtered out. Furthermore, tools that did not get their author's input in the quality assessment of the classification were also filtered out. In total, 23 tools remained for the classification. We were inspired by the work of Falcone et al. in setting up the classification table with the possible values for each data point [32]. The Excel sheet artifact, containing the extracted tools from the taxonomy paper, is also available in our git repository.

[1] `https://gitlab.informatik.hu-berlin.de/se/BA-Wagner-Eugen/-/tree/wagnereu-main-patch-65330/Data`

In the next step, we carefully reviewed and read the available resources about the tools in our set of 67 research papers. Out of those 67 papers, we were able to extract 47 RV tools, based on experiments, demonstrations, or general evaluations. In summary, with our 47 tool entries combined with the 23 tools from Falcone et al. [32], we took a set of 70 tools into the final consideration for the classification. Based on the classification table, we then extracted the following defined data points. We explain the possible values and abbreviations for each entry in the classification table.

List of possible values and abbreviations used for Table 1.

DP1 (Area of improvement):
- property verification (pv)
- failure prevention or reaction (fp)
- testing or debugging (td)
- information collection (ic)

DP2 (Proposed objective):
- scalability (sc)
- accessibility (acc)
- reactivity (r)
- overhead efficiency and performance (oe)
- correctness and accuracy (cs)

DP3 (Specification language/ formalism):
- <custom input>

DP4 Output:
- a single (sng) + verdict OR witness OR robustness
- a sequence of (seq) + verdict OR witness OR robustness
- graphical (g)

DP5 (Interference):
- invasive
- non-invasive

DP6 (Active Reaction):
- exception (ex)
- recovery (rc)
- rollback (ro)
- enforcement (en)

DP7 (Passive Reaction):
- specification output (so)
- explanation generation (exp)
- statistics (st)

DP8 (Stage):
- online (on)
- offline (off)

DP9 (Synchronisation):
- synchronous (sync)
- asynchronous (async)

General values:
- all values applicable (all)
- no values applicable (none)
- data point is not applicable (na)
- insufficient information (?)

Tool name	DP1	DP2	DP3	DP4	DP5	DP6	DP7	DP8	DP9	Reference
Aerial	pv,fp	cs	MTL, MDL	seq(v)	non-invasive	none	so	all	sync	[12, 32]
AllenRV	pv,td,ic	sc,r	LTL	seq(v,w)	non-invasive	none	so	all	sync	[83]
Alloy	pv,td	sc,oe	Alloy	sng(w)	non-invasive	none	exp	off	sync	[81]
AntidoteRT	pv, fp	oe	AntidoteRT specific	seq(v,w,r)	invasive	rc,en	exp	all	all	[80]
ARTiMon	pv,fp,td	cs,r	ARTiMon	seq(v)	invasive	?	so	all	sync	[64, 32]
AspectSol	pv,fp	r	Graph-based	sng(v)	non-invasive	en	none	on	sync	[3]
BeepBeep 3	pv,td,ic	sc,acc,oe	Lola, QEA, LTL-FO+	seq(v)	non-invasive	none	so,st	all	async	[41, 32]
BISM	ic	oe	BISM	seq(w)	invasive	en	so,st	all	async	[76]
Contract Larva	pv,fp	cs	DEA	sng(v)	invasive	na	na	on	sync	[29, 32]
Coq	pv,ic	cs	MTL	seq(w)	non-invasive	na	so,st	all	async	[26, 17]
CPSDebug	td	acc	STL	seq(w)	invasive	na	exp	off	na	[10, 32]
DecentMon	pv,td,ic	acc	LTL	sng(v)	non-invasive	na	so,st	off	na	[32]
DejaVu	pv,td	sc	QTL	seq(v,w)	all	none	so	all	all	[46, 32, 78]
detectEr	pv	r,cs	µHML	sng(v)	all	none	so	on	async	[32]
Diamont	pv	cs,oe	Diamont-IR	seq(v,r)	non-invasive	none	so	off	async	[34]
DR-BIP	pv,ic	acc	DR-BIP, LTL	seq(v,w,r)	non-invasive	none	so,st	all	sync	[28]
E-ACSL	pv,fp,td	sc,acc	E-ACSL	sng(v)	invasive	ex,rc	none	on	sync	[55, 32]
FalCAuN	pv,td	r,oe	LTL,STL	seq(w)	non-invasive	none	so,exp	on	sync	[74]
FRed, CPAChecker	pv,ic	sc,oe	Automata (CFA)	sng(w),g	non-invasive	none	so,st	all	sync	[15, 42]
GIFC	pv,fp,td	r,cs	GIFC specific	na	invasive	ex,en	so,exp	all	all	[71]
GREP	pv,fp	acc	Automata	sng(v),g	invasive	en	so	all	async	[32]
iCFTL (VyPR)	pv,ic	acc,cs,oe	CFTL	sng(v,w,r)	non-invasive	none	so,exp	all	sync	[25, 24]
Into the Unknown	fp	cs	Trained model	seq(v,r)	invasive	ro,en	exp	on	sync	[58]
Isabelle/HOL	pv,ic	cs,oe	IMP+pseq	seq(w)	non-invasive	none	so	off	sync	[49]
Larva	pv,ic	cs	DATEs	sng(v)	invasive	ex,rc	so	on	sync	[2, 32]
LogFire	pv,td,ic	sc,acc	LogFire DSL	sng(v,w)	invasive	none	so,exp	all	all	[32]
RDE	pv,td,ic	acc,oe,cs	Lola, RTLola	seq(v,w)	invasive	none	so,exp	on	sync	[54]
dLola	pv	oe	dLola, LTL	sng(v)	non-invasive	none	so	on	sync	[23]
MarQ	pv,td	cs	QEA	sng(v)	invasive	none	so	all	sync	[32]
MESA	pv,td,ic	sc,oe	TraceContract DSL	seq(w),g	non-invasive	none	all	on	sync	[73]
METIS	pv,fp	sc,oe	Automata	sng(v)	non-invasive	none	so,exp	on	sync	[1]
ModBat	td	acc,cs	Scala assertions	sng(v,w),g	non-invasive	none	all	on	sync	[32]
MonAmi	pv	oe	MFOTL	seq(v,w)	non-invasive	none	so	all	sync	[45]
MonPoly	pv,td,ic	acc,oe,cs	MFOTL, MTL	seq(v,w)	non-invasive	none	so,exp	all	all	[11, 68, 67]
Montre	ic	cs	TRE	seq(v,w)	non-invasive	none	so,exp	all	sync	[32]
MoonLight	pv,td	sc,oe	STREL, STL	sng(v,w)	non-invasive	none	so	off	async	[8]
nfer	pv,td,ic	acc	Automata	seq(v,w)	non-invasive	none	so	all	async	[32, 51]
NuRV	pv	acc, oe	LTL	seq(w)	non-invasive	none	so	all	sync	[19]
Orchids	pv,fp	r	Orchids specific	sng(v,w)	non-invasive	ex	so	all	async	[32]
Ortac	td	acc	Ocaml	seq(v,w)	non-invasive	none	so	on	sync	[35, 36]
OSIP	pv	sc,oe	Classifier (NN)	seq(v,w)	invasive	en,ro	none	on	sync	[44]
ParTraP	pv,td	acc,oe	ParTraP specific	sng(w)	non-invasive	none	exp,st	off	na	[13, 32]
PatrIoT	fp	cs,oe	QF-MTL	na	invasive	all	none	on	sync	[84]
PerceMon	pv,td	cs	SQTL, TQTL	seq(w,r),g	non-invasive	none	so,exp	on	sync	[6]
POMC	pv	oe,cs	POTL, MiniProc DSL	seq(w),sng(v,r)	non-invasive	none	so,exp	off	sync	[63]
Prevent	pv	oe,cs	DTMC, HMM	seq(w)	non-invasive	none	so	on	sync	[4, 5]
PropaFP	pv	cs	SPARK, Ada	seq(v,w)	non-invasive	none	so,exp	on	sync	[65]
PyContract	pv	acc,oe,cs	PyContract specific	sng(v)	non-invasive	none	so	?	sync	[22]
Reelay	pv,td	acc,oe,cs	MTL, RegEx	seq(v)	non-invasive	none	so	on	sync	[59, 32]
RML	pv,td	acc	RML	seq(v)	non-invasive	none	so	all	all	[32]
RSMCheck	pv,ic	sc,oe	CTL	seq(v,w)	non-invasive	none	so,exp,st	off	async	[27]
rtamt	td,ic	sc	STL, CARLA	seq(r)	non-invasive	none	so,st	on	all	[85]
RVHyper	pv	sc,acc	HyperLTL	seq(v,w)	all	none	so	off	sync	[40]
ShapeIt	ic	acc,cs	ShapeIt specific	seq(w)	non-invasive	na	na	off	sync	[9]
SharpDetect	ic	acc	Assembly (C#)	seq(w)	invasive	none	so	off	sync	[20]
SOLOIST-ZOT	pv	cs	SOLOIST	sng(v)	non-invasive	na	so	off	na	[32]
SOTER	all	acc,r,cs	SOTER specific (P)	all	invasive	en	so,st	on	all	[75]
S-TaLiRo	pv,td	acc,cs	MTL	seq(v,w,r)	non-invasive	none	so,exp,st	all	sync	[30]
STLGym	pv	cs	STL	na	non-invasive	en,rc	none	on	sync	[43]
Striver, HStriver	pv,td,ic	sc,acc,oe	Striver	seq(v),g	non-invasive	none	so	on	async	[39]
TACoS	pv,td	acc,oe	MTL, ATA	seq(v,w),g	non-invasive	none	so,st	all	sync	[48]
TeSSLa	pv	acc,r	TeSSLa specific	all	all	none	so,exp,st	all	async	[50]
THEMIS	pv,td,ic	sc,acc	LTL, Automata	seq(v)	non-invasive	ex	so,st	all	all	[32]
TiPEX	fp	cs	Automata	seq(v)	invasive	en	so	on	async	[33, 32]
TLTk	pv,ic	acc,oe	MTL,STL	seq(v,w,r)	invasive	en	so,st	on	sync	[21]
TraceContract	all	acc,cs	TraceContract DSL	sng(w)	invasive	ex,rc	all	all	all	[32]
UPPAAL	pv,td,ic	acc,oe	UPPAAL specific	all	non-invasive	none	all	on	sync	[57, 60, 53]
VerifAI	td,ic	acc,cs,oe	MTL, Lola, Scenic	seq(v,r)	non-invasive	none	all	on	na	[79, 82]
VeriMon	pv,td,ic	cs	MFOTL	seq(v,w)	non-invasive	none	so,exp	all	?	[69]
Viper	pv	cs	Lola, RTLola	seq(v,w)	non-invasive	none	so,st	on	all	[37]

Table 1: Classification of regarded RV tools

The possible values for data points DP1, DP4, DP5, DP6, DP7, DP8, and DP9 are inspired by or adopted unchanged from the taxonomy paper of Falcone et al. [32]. DP1, DP3, DP5, DP6, DP7, DP8, and DP9 have been adopted with minor renaming changes to the abbreviations. For DP4, we have adopted all values and included the value "graphical". This was necessary to include since the tools we've reviewed focused on visualizing the output data. The other values and data points do not reflect the option to represent such an entry, and therefore we deemed it necessary to include. DP2, with its values, was entirely created by us. We saw a clear need for a distinction between what a tool proposes to do and what research about said tool proposes to improve within the tool while reviewing each individual tool. The reason for that is the possibility to analyze research directions for certain tools (e.g. nfer).

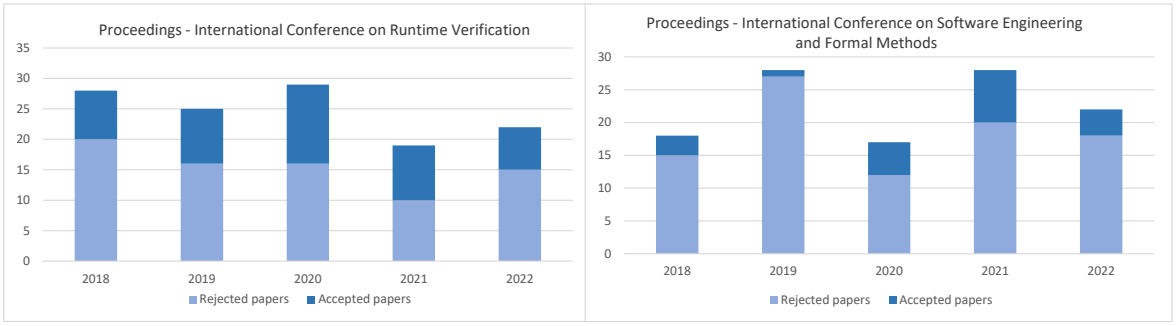

Figure 4: Overview of the quantity of filtered research papers from the conference proceedings per year of the RV conference (left) and the SEFM conference (right).

For the classification table, each data point represents a row next to the tool name and references. The final classification table contains 70 distinct tool entries (see Table 1). We have excluded repeated tool entries from the table for the sake of readability (e.g. MonPoly [11, 68, 67]). For all tool entries, we attempted to provide clickable URLs containing the tool's official homepage or a code repository containing the experimentation results from the papers we reviewed. We were unable to find homepages or git repositories for the following tools: CPSDebug [10, 32], Diamont [34], MESA [73], METIS [1], MonAmi [45], NuRV [19], OSIP [44], PyContract [22], RVHyper [40] and VeriMon [69]. For this reason, these tools do not contain a clickable URL.

The toolbox CADP was extended in order to ease the understanding of counterexamples [7]. When a given model representing a specification does not accept a trace, a counterexample aims to give insight into why the trace did not satisfy the specification. This functions as an explanation. However, no git repository or website was provided containing further information about the tool's functionality [7]. While CADP may fit the data filtering criteria, we could not find other research papers in the regarded time frame or the defined search domains utilizing it. Therefore, it was excluded from the

classification.

The tools FRed [15] and CPAChecker [42] were combined into one entry due to FRed being classified as an extension of CPAChecker. Both tools share many use cases and were therefore classified as one entry.

Next, we discuss the threats to validity from our classification results. While we have already discussed the threats to validity of the mapping study process in Section 3.1., there are other points we wish to discuss in regards to our classification results. On one hand, we were primarily able to extract information about the tools directly from the research papers. Secondarily, we looked up documentation in available git repositories or websites for the tools. Unlike the work of Falcone et al. [32], we are unable to verify the results by conducting a questionnaire. Furthermore, the documentation on some tools was broad or specifically created to evaluate or demonstrate the research results. One such example was the tool GRACE, which we decided to exclude from the classification due to being in an experimental stage [18]. The decision procedure for determining whether a tool would be applicable to use cases outside of experimentation is purely qualitative and may be biased.

4.2 Evaluation

In the following section, we discuss how the classification results correlate to and answer the research questions. We created the data points in correlation with the research questions. We explicitly talk about and discuss the classified tools instead of the research papers behind them. To review the references for each tool, we refer to Table 1. DP1 and DP2 answer RQ1. DP3 answers RQ2. DP4 yields answers to RQ5. DP6 and DP7 answer RQ3, and lastly, DP5, DP8, and DP9 answer RQ4 (see Table 2).

Data point	Research question
DP1	RQ1
DP2	RQ1
DP3	RQ2
DP4	RQ5
DP5	RQ4
DP6	RQ3
DP7	RQ3
DP8	RQ4
DP9	RQ4

Table 2: Our data points in relation to our research questions.

In Figure 5, we visualize the quantities of area of improvement (DP1) and proposed research objective (DP2). Entries in the area of improvement and the proposed objective are, of course, not mutually exclusive. This means that any tool may have multiple classified areas of improvement or research objectives.

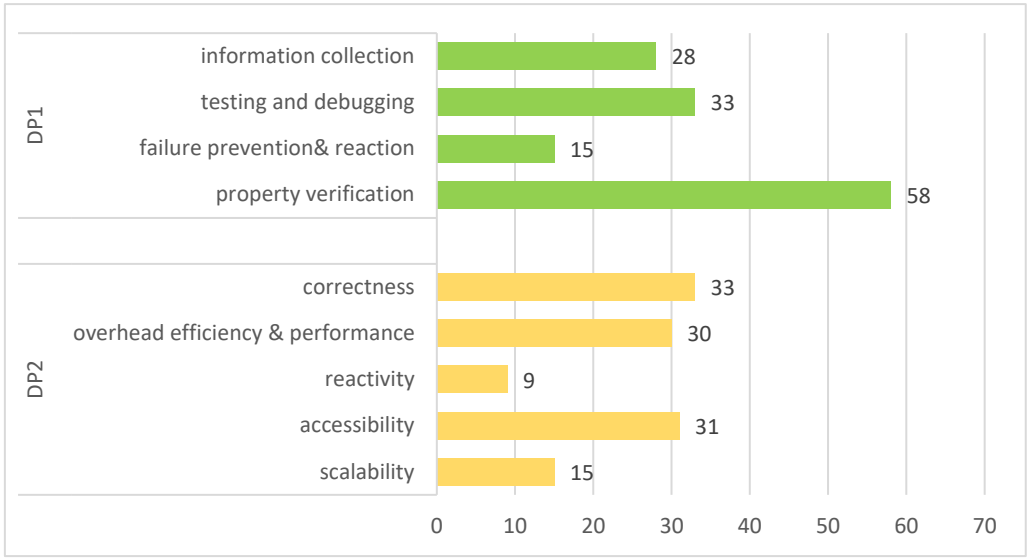

Figure 5: Overview of different areas of improvement (top) of the tools (DP1) and different proposed objectives (bottom) within the papers (DP2).

For DP1, the most used area of improvement is property verification, with 58 out of 70 tools being classified as such. Examples of primarily property-verifying tools include Diamont [34], dLola [23], NuRV [19] and RVHyper [40]. For testing and debugging, we have classified 33 out of 70 tools. Filliâtre et al. presented the tool Ortac to implement runtime assertion checking with a testing focus [35]. In our classification, 28 out of 70 tools were deemed to focus on information collection within the system they're monitoring. The tools BISM [76], SharpDetect [20] and ShapeIt [9] are strictly improving the information collection aspect. With failure prevention and reaction, we have classified the least amount of tools at 15 out of 70. One such tool is PatrIoT [84], which is able to detect policy violations and effectively enforce control actions in order to combat them.

With DP2, we analyzed the references behind the classified tools for proposed research questions and goals. Based on our analysis, we then created the values for DP2. This was done for cases where RV tools have already been developed and used, but research is heading in a certain direction with the goal of extending to another area for the application of the tool. A total of 33 out of 70 tools were classified as correctness (and accuracy). Tools such as STLGym [43] , Viper [37], and Into the Unknown [58] focus entirely on delivering accurate results in discovering faulty traces. With accessibility, we counted the research focus of making the tool usable and deployable by non-experts in the field of RV. We counted 31 out of 70 tools. A fitting example is Ortac, yet again [35]. The tool contains a manageable complexity of deployment steps and usage in the

well-documented ocaml-gospel specification language. 30 out of 70 tools were classified as having the research goal of improving overhead efficiency and performance. The tool RSMCheck by Dubslaff et al. promises a faster data-flow analysis approach compared to state-of-the-art model-checking tools [27]. Regarding scalability over growing system architectures, 15 out of 70 were classified as such. The tools DejaVu [46, 32, 78] and rtamt [85] were solely classified as being aimed towards a growing scale. Lastly, 9 out of 70 tools were classified as being geared towards the reactivity of errors within the system. The tool Orchids [32] is aimed at discovering intrusions within logfiles in real time and reacting to them.

Now we can answer RQ1 (What is the desired and achieved goal of the tools?) with the information from DP1 and DP2. On one hand, the most prominent goal achieved by the classified tools is to verify a given property in trace samples. On the other hand, the most experimented-on goal is to improve or prove the correctness of the RV monitor within the tool.

With DP3, we looked at the formal language that was used in order to input specifications. We classified 32 out of 70 tools as grammar rewrite-based with LTL or a derivative language of LTL (e.g. STL, MTL, etc.). AllenRV uses LTL with metric extensions to model past and future properties [83]. The second group of formalisms was categorized as "specific" to the tool and not directly applicable to other tools. We classified 26 tools out of 70 as tool-specific. It's important to note that tool-specific formalisms can also be based on LTL or automata. One such example is the tool ShapeIt [9]. ShapeIt is able to mine specifications from the target domain (CPS) in a five-step process, generating logic formula fragments and utilizing automata-learning to output regular expressions. Lastly, 12 out of 70 tools were primarily based on model-checking approaches using automata. The tool METIS is primarily utilizing automata to model properties with the goal of reducing redundancies in the monitor [1].

Regarding RQ2 (Which specification formalism is most commonly used in RV tools?) we come to the conclusion that grammar-based approaches with LTL are the most commonly used specification formalisms to model specifications.

With DP6 and DP7, we reviewed the way the tools react to discovered undesired behavior within the system. In Figure 6, we visualize our results of each value's count. In regards to active reactions, we classified 12 out of 70 tools as containing runtime enforcement features. The tools AntidoteRT [80], Into the Unknown [58] and OSIP [44] serve as examples of tools containing enforcement reactions. An interesting observation is that all three listed tools are designed for adaptive systems, using enforcement to influence processing steps within the neural network. 7 out of 70 tools were classified as containing exception steps. The tool Orchids contains steps for exception handling in the implementation [32]. We classified 6 out of 70 tools as having recovery mechanisms. And lastly, 3 out of 70 tools contained rollback features to revert the system to a state before the error occurred. OSIP [44] and Into the Unknown [58] are examples for tools containing this feature. The most prevalent value for active reaction however, was "none" with 46 out of 70 tools. One example of a tool without an active reaction is Ortac [35]. Lastly, 6 tools had a very specific architecture and contained countless

parts, which we classified as "not applicable". One such tool is Coq [26, 17]. Coq is a proof assistant that, as a base tool, is not required to be applied to a system and therefore can't react actively. With the tool ARTiMon [55, 32], we were unable to find enough information to classify the active reaction aspect.

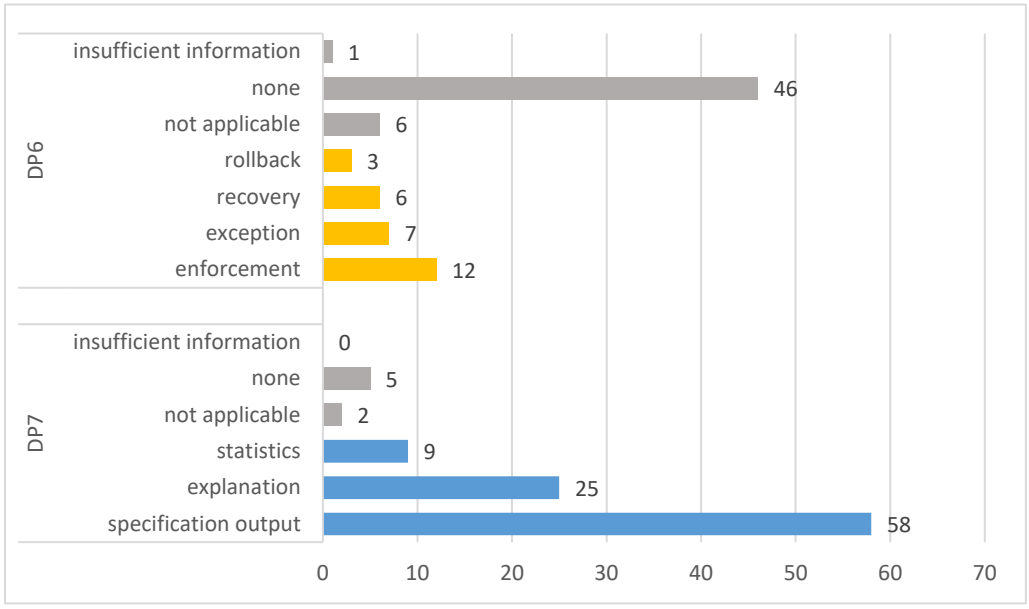

Figure 6: Overview of the tools' reaction. Please note that a tool may have an active reaction and a passive reaction approach (DP6, DP7).

In DP7, we reviewed the passive reaction aspect of the tools. The most prevalent value was "specification output", meaning most tools output their findings of the trace sample after processing. The tools RVHyper [40], RSMCheck [27] and MoonLight [8] are a few examples of having a specification output. 25 out of 70 tools react passively by outputting an explanation for why certain verdicts were reached with regards to the specification analysis. The tools Alloy [81] and CPSDebug [10] primarily focus on outputting an explanation. 9 out of 70 tools offer additional information in the form of statistics about the quantity of violated or satisfied specifications [32]. The framework TeSSLa [50] and the tool S-TALIRO [30] were classified as containing statistics among their features. Lastly, 5 out of 70 tools did not contain a passive reaction, and 2 tools were not applicable for the same reasons as with the active reaction. No tools contained insufficient information in regards to a passive reaction.

Now we can answer RQ3 (How do the tools handle discovered errors?). We can answer this question twofold. First, in regards to a passive reaction, the majority of tools output the results of the specification analysis. Second, in regards to an active reaction, the majority of tools do not contain features that can be classified as active

reactions.

In DP5, we reviewed the qualitative measure of invasiveness of the tools (see Figure 7. On one hand, we classified 50 tools as primarily non-invasive, meaning they do not share memory space or measure components within the system that the system accesses as well. On the other hand, 24 out of 70 tools were deemed primarily invasive. The tools DejaVu [46, 32, 78], detectEr [32], RVHyper [40] and TeSSLa [50] had a more complex architecture with both invasive and non-invasive features.

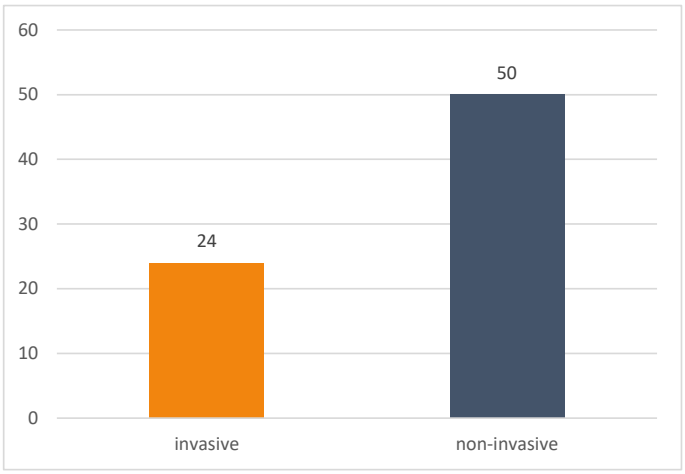

Figure 7: Overview of the amount of tools classified as invasive against non-invasive (DP5). Tools can be classified as both values if they contain architectures with invasive and non-invasive parts.

For DP8 and DP9, we looked at when the monitor operates in regards to the system and how it is synchronized time-wise. In Figure 8, we visualize the counting of tools in regards to DP8. We classified 56 tools as having an online monitoring component, meaning the monitor continuously runs during the system's runtime. The tool PATRIOT [84] contains an exclusively online monitoring component to actively enforce policy violations in IoT systems. Furthermore, we classified 41 out of 70 tools as having an offline monitoring component. This means that the monitor runs outside of the system's runtime. The proof assistant Isabelle [49] and the tool MoonLight [8] both exclusively execute offline. A total of 28 out of 70 tools contained either a monitor that can run both in offline and online mode or multiple monitors with a task separation covering online and offline. The tool PyContract had insufficient information to fulfill this data point [22].

Regarding the synchronization of the tool's runtime, we classified 51 out of 70 tools as running at least partly in synchronization with the system (see Figure 9). Examples for synchronous runtime include the tools SharpDetect [20], TLTk [21] and UPPAAL [57, 60, 53]. Furthermore, 24 out of 70 tools were classified as running asynchronously to the system (e.g. nfer [32, 51], Striver [39], TiPEX [33, 32]). 11 out of 70 tools were

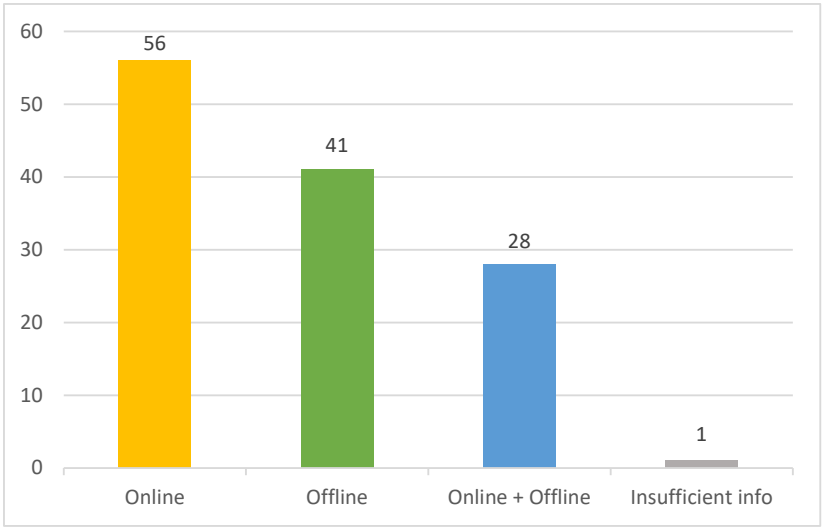

Figure 8: Overview of the deployment of RV monitors within the classified tools (DP8).

classified as containing both synchronous and asynchronous parts in their architecture. With even more complex architectures, we deemed the tools as not applicable in regards to synchronization. Examples include the tools VerifAI [79, 82] and ParTraP [13, 32]. We were unable to find enough information to classify the tool VeriMon [69] in regards to synchronization and therefore classified it as "insufficient information".

With the information from DP5, DP8 and DP9 in mind, we can answer RQ4 (How are the tools deployed?). First, we have seen the majority of tools use non-invasive approaches in regards to interference with the regarded system. Second, the majority of RV monitors are operating online during the runtime of the regarded system. However, offline monitors are also frequently used. Third, the majority of tools operate synchronously without requiring a system halt.

Regarding the final data point DP4, we looked at the way the tools output their findings. In Figure 10, we visualize the counting of the individual possible values. On one hand, 47 out of 70 tools use either a sequence of verdicts, witnesses, or robustnesses. On the other hand, 22 out of 70 tools use a single output value of either a verdict, a witness or a robustness. 7 out of 70 tools use visualization features to generate graphical output. Examples include the tools PerceMon [6] or TACoS [48]. We classified 3 tools as not being applicable for the output classification in the case of the tool's functionality being too specific to the applied system (e.g. GIFC [71]).

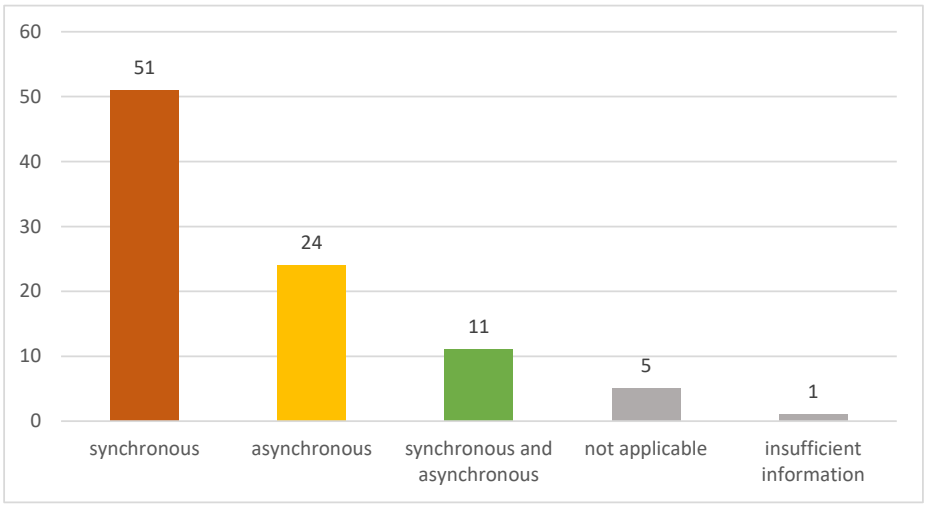

Figure 9: Overview of the synchronization approach of the RV monitor (DP9).

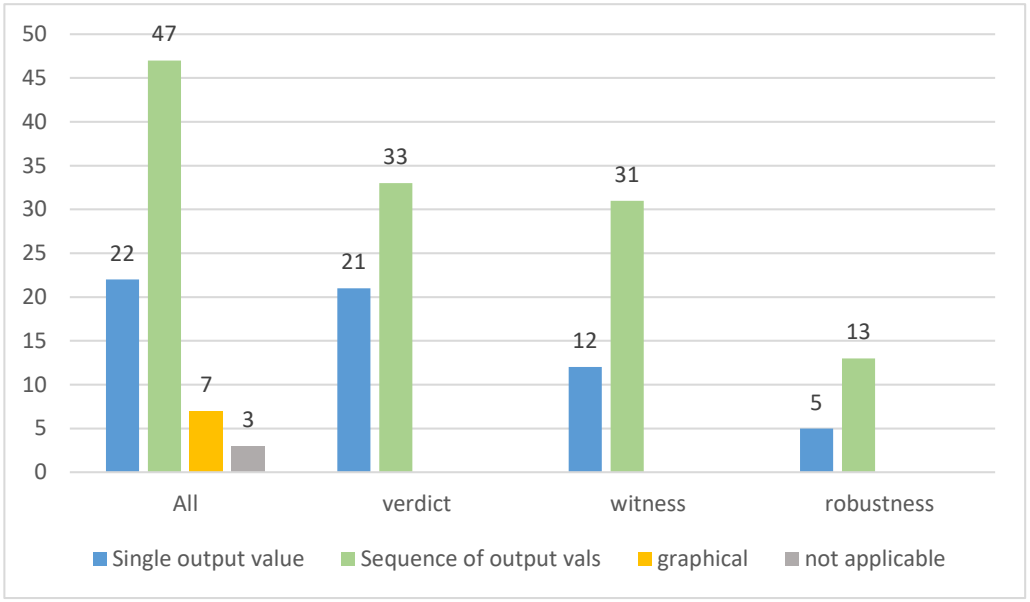

Figure 10: Overview of the output values (DP4). On the very left, the amount of each output structure is depicted. The three other bar groups depict the amount of single or sequence output approaches per value (verdict, witness, robustness). The values are not mutually exclusive.

Out of 22 tools using a single output structure for the output, 21 output a single verdict. Examples include E-ACSL [55, 32], Larva [29, 32] and MoonLight [8]. Furthermore 12 out of 22 tools output a single witness (e.g. MoonLight). Finally, 5 out of 22 tools output a single robustness value, like the tool iCFTL built on the VyPR framework [25, 24]. These values are not mutually exclusive. This means that, for example, MoonLight can output both a single verdict and a single witness.

Out of 47 tools, using a sequence of output values, 33 tools output a sequence of verdicts. Examples include Viper [37], Reelay [59, 32] and PropaFP [65]. Furthermore, 31 tools in this set output a sequence of witnesses, like DR-BIP [28] or MonPoly [11, 68, 67]. Finally, 13 out of 47 tools that output a sequence of values used robustness values as output. Tools like Diamont [34], AntidoteRT [80] and VerifAI [79, 82] attempt to measure the degree of violation that occurred on given specifications. These values are also not mutually exclusive. A tool may have a sequence of verdicts and witnesses, in addition to a single robustness output, bundled with a graphical visualization.

With this information in mind, we can answer the final research question RQ5 (How are the tools evaluating the trace?). We conclude that the majority of tools output sequences of values. In most cases, these sequences contain only verdicts. However, witnesses are also often used for a sequence of output values.

5 Conclusion

In this work, we have conducted a systematic mapping study on RV tools for the time span of 2016-2022. We adopted parts of the taxonomy introduced by Falcone et al. [32] and applied data filtering criteria to their set of classified tools. In total, we adopted 23 classified tool entries from their work. Their tool entries covered the time span from 2016 to 2018. Then we added 47 RV tool entries to the set and classified them. The entries were extracted from the time span of 2018 to 2022 from the SEFM and RV conferences.

We have shown the primary design choices for the applicability of recently developed or extended RV tools. Research in the form of surveys on tools is rare in the field of RV. This leads to frequent research gaps in the contextualization of developments. With this work, we contribute a tabular collection of 70 RV tools and their classification.

For future work, we consider adding a quality assessment to our classification table by reaching out to the tool's authors to verify our results. Furthermore, we believe it is important to test the tools on a domain that has been specifically set up for demonstration runs. This, however, is a time-consuming research direction, as many of our mentioned tools have specific input or deployment requirements. We also recommend adding other research platforms to the search domains to look for further RV tool research.

References

[1] G. Allabadi, A. Dhar, A. Bashir, and R. Purandare. METIS: Resource and context-aware monitoring of finite state properties. In C. Colombo and M. Leucker, editors, *Runtime Verification*, pages 167–186, Cham, 2018. Springer International Publishing.

[2] A. Aranda García, M.-E. Cambronero, C. Colombo, L. Llana, and G. J. Pace. Runtime verification of contracts with Themulus. In F. de Boer and A. Cerone, editors, *Software Engineering and Formal Methods*, pages 231–246, Cham, 2020. Springer International Publishing.

[3] S. Azzopardi, J. Ellul, R. Falzon, and G. J. Pace. AspectSol: A solidity aspect-oriented programming tool with applications in runtime verification. In T. Dang and V. Stolz, editors, *Runtime Verification*, pages 243–252, Cham, 2022. Springer International Publishing.

[4] R. Babaee, A. Gurfinkel, and S. Fischmeister. Predictive run-time verification of discrete-time reachability properties in black-box systems using trace-level abstraction and statistical learning. In C. Colombo and M. Leucker, editors, *Runtime Verification*, pages 187–204, Cham, 2018. Springer International Publishing.

[5] R. Babaee, A. Gurfinkel, and S. Fischmeister. Prevent: A predictive run-time verification framework using statistical learning. In E. B. Johnsen and I. Schaefer, editors, *Software Engineering and Formal Methods*, pages 205–220, Cham, 2018. Springer International Publishing.

[6] A. Balakrishnan, J. Deshmukh, B. Hoxha, T. Yamaguchi, and G. Fainekos. PerceMon: Online monitoring for perception systems. In L. Feng and D. Fisman, editors, *Runtime Verification*, pages 297–308, Cham, 2021. Springer International Publishing.

[7] G. Barbon, V. Leroy, and G. Salaün. Counterexample simplification for liveness property violation. In E. B. Johnsen and I. Schaefer, editors, *Software Engineering and Formal Methods*, pages 173–188, Cham, 2018. Springer International Publishing.

[8] E. Bartocci, L. Bortolussi, M. Loreti, L. Nenzi, and S. Silvetti. Moonlight: A lightweight tool for monitoring spatio-temporal properties. In *Runtime Verification*, pages 417–428. Springer International Publishing, Oct. 2020.

[9] E. Bartocci, J. Deshmukh, C. Mateis, E. Nesterini, D. Ničković, and X. Qin. Mining shape expressions with ShapeIt. In R. Calinescu and C. S. Păsăreanu, editors, *Software Engineering and Formal Methods*, pages 110–117, Cham, 2021. Springer International Publishing.

[10] E. Bartocci, N. Manjunath, L. Mariani, C. Mateis, and D. Ničković. Automatic failure explanation in CPS Models. In P. C. Ölveczky and G. Salaün, editors, *Software Engineering and Formal Methods*, pages 69–86, Cham, 2019. Springer International Publishing.

[11] D. Basin, M. Gras, S. Krstić, and J. Schneider. Scalable online monitoring of distributed systems. In J. Deshmukh and D. Ničković, editors, *Runtime Verification*, pages 197–220, Cham, 2020. Springer International Publishing.

[12] D. Basin, S. Krstić, and D. Traytel. Almost event-rate independent monitoring of metric dynamic logic. In S. Lahiri and G. Reger, editors, *Runtime Verification*, pages 85–102, Cham, 2017. Springer International Publishing.

[13] A. Ben Cheikh, Y. Blein, S. Chehida, G. Vega, Y. Ledru, and L. du Bousquet. An environment for the ParTraP trace property language (tool demonstration). In C. Colombo and M. Leucker, editors, *Runtime Verification*, pages 437–446, Cham, 2018. Springer International Publishing.

[14] S. Berkovich, B. Bonakdarpour, and S. Fischmeister. GPU-based runtime verification. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1025–1036, 2013.

[15] D. Beyer and M.-C. Jakobs. FRed: Conditional model checking via reducers and folders. In F. de Boer and A. Cerone, editors, *Software Engineering and Formal Methods*, pages 113–132, Cham, 2020. Springer International Publishing.

[16] M. Bures, M. Klima, V. Rechtberger, X. Bellekens, C. Tachtatzis, R. Atkinson, and B. S. Ahmed. Interoperability and integration testing methods for iot systems: A systematic mapping study. In F. de Boer and A. Cerone, editors, *Software Engineering and Formal Methods*, pages 93–112, Cham, 2020. Springer International Publishing.

[17] A. Chattopadhyay and K. Mamouras. A verified online monitor for metric temporal logic with quantitative semantics. In J. Deshmukh and D. Ničković, editors, *Runtime Verification*, pages 383–403, Cham, 2020. Springer International Publishing.

[18] A. Cimatti, C. Tian, and S. Tonetta. Assumption-based runtime verification with partial observability and resets. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification*, pages 165–184, Cham, 2019. Springer International Publishing.

[19] A. Cimatti, C. Tian, and S. Tonetta. NuRV: A nuXmv extension for runtime verification. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification*, pages 382–392, Cham, 2019. Springer International Publishing.

[20] A. Čižmárik and P. Parízek. SharpDetect: Dynamic analysis framework for C#/.NET programs. In J. Deshmukh and D. Ničković, editors, *Runtime Verification*, pages 298–309, Cham, 2020. Springer International Publishing.

[21] J. Cralley, O. Spantidi, B. Hoxha, and G. Fainekos. TLTk: A toolbox for parallel robustness computation of temporal logic specifications. In J. Deshmukh and D. Ničković, editors, *Runtime Verification*, pages 404–416, Cham, 2020. Springer International Publishing.

[22] D. Dams, K. Havelund, and S. Kauffman. A python library for trace analysis. In T. Dang and V. Stolz, editors, *Runtime Verification*, pages 264–273, Cham, 2022. Springer International Publishing.

[23] L. M. Danielsson and C. Sánchez. Decentralized stream runtime verification. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification*, pages 185–201, Cham, 2019. Springer International Publishing.

[24] J. H. Dawes and D. Bianculli. Specifying properties over inter-procedural, source code level behaviour of programs. In L. Feng and D. Fisman, editors, *Runtime Verification*, pages 23–41, Cham, 2021. Springer International Publishing.

[25] J. H. Dawes and G. Reger. Explaining violations of properties in control-flow temporal logic. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification*, pages 202–220, Cham, 2019. Springer International Publishing.

[26] C. Dubois, O. Grinchtein, J. Pearson, and M. Carlsson. Exploring properties of a telecommunication protocol with message delay using interactive theorem prover. In E. B. Johnsen and I. Schaefer, editors, *Software Engineering and Formal Methods*, pages 239–253, Cham, 2018. Springer International Publishing.

[27] C. Dubslaff, P. Wienhöft, and A. Fehnker. Be lazy and don't care: Faster CTL model checking for recursive state machines. In R. Calinescu and C. S. Păsăreanu, editors, *Software Engineering and Formal Methods*, pages 332–350, Cham, 2021. Springer International Publishing.

[28] A. El-Hokayem, S. Bensalem, M. Bozga, and J. Sifakis. A layered implementation of DR-BIP supporting run-time monitoring and analysis. In F. de Boer and A. Cerone, editors, *Software Engineering and Formal Methods*, pages 284–302, Cham, 2020. Springer International Publishing.

[29] J. Ellul and G. J. Pace. Runtime verification of ethereum smart contracts. In *2018 14th European Dependable Computing Conference (EDCC)*, pages 158–163, 2018.

[30] G. Fainekos, B. Hoxha, and S. Sankaranarayanan. Robustness of specifications and its applications to falsification, parameter mining, and runtime monitoring with S-TaLiRo. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification*, pages 27–47, Cham, 2019. Springer International Publishing.

[31] Y. Falcone, S. Krstić, G. Reger, and D. Traytel. A taxonomy for classifying runtime verification tools. In C. Colombo and M. Leucker, editors, *Runtime Verification*, pages 241–262, Cham, 2018. Springer International Publishing.

[32] Y. Falcone, S. Krstić, G. Reger, et al. A taxonomy for classifying runtime verification tools. *International Journal on Software Tools for Technology Transfer*, 23:255–284, May 2021.

[33] Y. Falcone and S. Pinisetty. On the runtime enforcement of timed properties. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification*, pages 48–69, Cham, 2019. Springer International Publishing.

[34] V. Fernando, K. Joshi, J. Laurel, and S. Misailovic. Diamont: Dynamic monitoring of uncertainty for distributed asynchronous programs. In L. Feng and D. Fisman, editors, *Runtime Verification*, pages 184–206, Cham, 2021. Springer International Publishing.

[35] J.-C. Filliâtre and C. Pascutto. Ortac: Runtime assertion checking for OCaml (tool paper). In L. Feng and D. Fisman, editors, *Runtime Verification*, pages 244–253, Cham, 2021. Springer International Publishing.

[36] J.-C. Filliâtre and C. Pascutto. Optimizing prestate copies in runtime verification of function postconditions. In T. Dang and V. Stolz, editors, *Runtime Verification*, pages 85–104, Cham, 2022. Springer International Publishing.

[37] B. Finkbeiner, S. Oswald, N. Passing, and M. Schwenger. Verified rust monitors for Lola specifications. In J. Deshmukh and D. Ničković, editors, *Runtime Verification*, pages 431–450, Cham, 2020. Springer International Publishing.

[38] H. J. Goldsby, B. H. C. Cheng, and J. Zhang. Amoeba-rt: Run-time verification of adaptive software. In H. Giese, editor, *Models in Software Engineering*, pages 212–224, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[39] F. Gorostiaga and C. Sánchez. Striver: Stream runtime verification for real-time event-streams. In C. Colombo and M. Leucker, editors, *Runtime Verification*, pages 282–298, Cham, 2018. Springer International Publishing.

[40] C. Hahn. Algorithms for monitoring hyperproperties. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification*, pages 70–90, Cham, 2019. Springer International Publishing.

[41] S. Hallé and R. Khoury. Writing domain-specific languages for BeepBeep. In C. Colombo and M. Leucker, editors, *Runtime Verification*, pages 447–457, Cham, 2018. Springer International Publishing.

[42] J. Haltermann and H. Wehrheim. Information exchange between over- and underapproximating software analyses. In B.-H. Schlingloff and M. Chai, editors, *Software Engineering and Formal Methods*, pages 37–54, Cham, 2022. Springer International Publishing.

[43] N. Hamilton, P. K. Robinette, and T. T. Johnson. Training agents to satisfy timed and untimed signal temporal logic specifications with reinforcement learning. In B.-H. Schlingloff and M. Chai, editors, *Software Engineering and Formal Methods*, pages 190–206, Cham, 2022. Springer International Publishing.

[44] V. Hashemi, P. Kouvaros, and A. Lomuscio. OSIP: Tightened bound propagation for the verification of ReLU neural networks. In R. Calinescu and C. S. Păsăreanu, editors, *Software Engineering and Formal Methods*, pages 463–480, Cham, 2021. Springer International Publishing.

[45] K. Havelund, M. Omer, and D. Peled. Monitoring first-order interval logic. In R. Calinescu and C. S. Păsăreanu, editors, *Software Engineering and Formal Methods*, pages 66–83, Cham, 2021. Springer International Publishing.

[46] K. Havelund and D. Peled. An extension of LTL with rules and its application to runtime verification. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification*, pages 239–255, Cham, 2019. Springer International Publishing.

[47] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24:189–215, 2004.

[48] T. Hofmann and S. Schupp. TACoS: A tool for mtl controller synthesis. In R. Calinescu and C. S. Păsăreanu, editors, *Software Engineering and Formal Methods*, pages 372–379, Cham, 2021. Springer International Publishing.

[49] J. J. Huerta y Munive. Affine systems of ODEs in Isabelle/HOL for hybrid-program verification. In F. de Boer and A. Cerone, editors, *Software Engineering and Formal Methods*, pages 77–92, Cham, 2020. Springer International Publishing.

[50] H. Kallwies, M. Leucker, M. Schmitz, A. Schulz, D. Thoma, and A. Weiss. TeSSLa – an ecosystem for runtime verification. In T. Dang and V. Stolz, editors, *Runtime Verification*, pages 314–324, Cham, 2022. Springer International Publishing.

[51] S. Kauffman. nfer – a tool for event stream abstraction. In R. Calinescu and C. S. Păsăreanu, editors, *Software Engineering and Formal Methods*, pages 103–109, Cham, 2021. Springer International Publishing.

[52] B. Kitchenham, O. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. Systematic literature reviews in software engineering – a systematic literature review. *Information and Software Technology*, 51:7–15, Jan. 2009.

[53] P. Kobialka, S. L. Tapia Tarifa, G. R. Bergersen, and E. B. Johnsen. Weighted games for user journeys. In B.-H. Schlingloff and M. Chai, editors, *Software Engineering and Formal Methods*, pages 253–270, Cham, 2022. Springer International Publishing.

[54] M. A. Köhl, H. Hermanns, and S. Biewer. Efficient monitoring of real driving emissions. In C. Colombo and M. Leucker, editors, *Runtime Verification*, pages 299–315, Cham, 2018. Springer International Publishing.

[55] N. Kosmatov, F. Maurica, and J. Signoles. Efficient runtime assertion checking for properties over mathematical numbers. In J. Deshmukh and D. Ničković, editors, *Runtime Verification*, pages 310–322, Cham, 2020. Springer International Publishing.

[56] P. Kushwaha, R. Purandare, and M. B. Dwyer. Optimal finite-state monitoring of partial traces. In *Runtime Verification*, pages 124–142. Springer International Publishing, Sept. 2022.

[57] L. Lestingi, M. Askarpour, M. M. Bersani, and M. Rossi. Formal verification of human-robot interaction in healthcare scenarios. In F. de Boer and A. Cerone, editors, *Software Engineering and Formal Methods*, pages 303–324, Cham, 2020. Springer International Publishing.

[58] A. Lukina, C. Schilling, and T. A. Henzinger. Into the Unknown: Active monitoring of neural networks. In L. Feng and D. Fisman, editors, *Runtime Verification*, pages 42–61, Cham, 2021. Springer International Publishing.

[59] K. Mamouras, A. Chattopadhyay, and Z. Wang. A compositional framework for quantitative online monitoring over continuous-time signals. In L. Feng and D. Fisman, editors, *Runtime Verification*, pages 142–163, Cham, 2021. Springer International Publishing.

[60] L. Miedema and C. Grelck. Strategy switching: Smart fault-tolerance for weakly-hard resource-constrained real-time applications. In B.-H. Schlingloff and M. Chai, editors, *Software Engineering and Formal Methods*, pages 129–145, Cham, 2022. Springer International Publishing.

[61] A. Momtaz, N. Basnet, H. Abbas, and B. Bonakdarpour. Predicate monitoring in distributed cyber-physical systems. In L. Feng and D. Fisman, editors, *Runtime Verification*, pages 3–22, Cham, 2021. Springer International Publishing.

[62] K. Petersen, S. Vakkalanka, and L. Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, 2015.

[63] F. Pontiggia, M. Chiari, and M. Pradella. Verification of programs with exceptions through operator precedence automata. In R. Calinescu and C. S. Păsăreanu, editors, *Software Engineering and Formal Methods*, pages 293–311, Cham, 2021. Springer International Publishing.

[64] N. Rapin. Reactive property monitoring of hybrid systems with aggregation. In Y. Falcone and C. Sánchez, editors, *Runtime Verification*, pages 447–453, Cham, 2016. Springer International Publishing.

[65] J. Rasheed and M. Konečný. Auto-active verification of floating-point programs via nonlinear real provers. In B.-H. Schlingloff and M. Chai, editors, *Software Engineering and Formal Methods*, pages 20–36, Cham, 2022. Springer International Publishing.

[66] G. Reger and K. Havelund. What is a trace? a runtime verification perspective. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, pages 339–355, Cham, 2016. Springer International Publishing.

[67] J. Schneider. Randomized first-order monitoring with hashing. In T. Dang and V. Stolz, editors, *Runtime Verification*, pages 3–24, Cham, 2022. Springer International Publishing.

[68] J. Schneider, D. Basin, F. Brix, S. Krstić, and D. Traytel. Scalable online first-order monitoring. In C. Colombo and M. Leucker, editors, *Runtime Verification*, pages 353–371, Cham, 2018. Springer International Publishing.

[69] J. Schneider, D. Basin, S. Krstić, and D. Traytel. A formally verified monitor for metric first-order temporal logic. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification*, pages 310–328, Cham, 2019. Springer International Publishing.

[70] M. Schwenger. Monitoring cyber-physical systems: From design to integration. In J. Deshmukh and D. Ničković, editors, *Runtime Verification*, pages 87–106, Cham, 2020. Springer International Publishing.

[71] A. L. Scull Pupo, L. Christophe, J. Nicolay, C. de Roover, and E. Gonzalez Boix. Practical information flow control for web applications. In C. Colombo and M. Leucker, editors, *Runtime Verification*, pages 372–388, Cham, 2018. Springer International Publishing.

[72] J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S. Smolka, S. Stoller, and E. Zadok. Interaspect: aspect-oriented instrumentation with GCC. In *Formal Methods in System Design*, volume 41, page 295–320, 2012.

[73] N. Shafiei, K. Havelund, and P. Mehlitz. Actor-based runtime verification with MESA. In J. Deshmukh and D. Ničković, editors, *Runtime Verification*, pages 221–240, Cham, 2020. Springer International Publishing.

[74] J. Shijubo, M. Waga, and K. Suenaga. Efficient black-box checking via model checking with strengthened specifications. In L. Feng and D. Fisman, editors, *Runtime Verification*, pages 100–120, Cham, 2021. Springer International Publishing.

[75] S. Shivakumar, H. Torfah, A. Desai, and S. A. Seshia. SOTER on ROS: A run-time assurance framework on the robot operating system. In J. Deshmukh and D. Ničković, editors, *Runtime Verification*, pages 184–194, Cham, 2020. Springer International Publishing.

[76] C. Soueidi, A. Kassem, and Y. Falcone. BISM: Bytecode-level instrumentation for software monitoring. In J. Deshmukh and D. Ničković, editors, *Runtime Verification*, pages 323–335, Cham, 2020. Springer International Publishing.

[77] C. Sánchez, G. Schneider, W. Ahrendt, E. Bartocci, D. Bianculli, C. Colombo, Y. Falcone, A. Francalanza, S. Krstić, J. M. Lourenço, D. Nickovic, G. J. Pace, J. Rufino, J. Signoles, D. Traytel, and A. Weiss. A survey of challenges for runtime verification from advanced application domains (beyond software). In *Formal Methods in System Design*, volume 54, pages 279–335, 2019.

[78] A. Temperekidis, N. Kekatos, and P. Katsaros. Runtime verification for FMI-based co-simulation. In T. Dang and V. Stolz, editors, *Runtime Verification*, pages 304–313, Cham, 2022. Springer International Publishing.

[79] H. Torfah, S. Junges, D. J. Fremont, and S. A. Seshia. Formal analysis of AI-based autonomy: From modeling to runtime assurance. In L. Feng and D. Fisman, editors, *Runtime Verification*, pages 311–330, Cham, 2021. Springer International Publishing.

[80] M. Usman, D. Gopinath, Y. Sun, and C. S. Păsăreanu. Rule-based runtime mitigation against poison attacks on neural networks. In T. Dang and V. Stolz, editors, *Runtime Verification*, pages 67–84, Cham, 2022. Springer International Publishing.

[81] C. Vick, E. Kang, and S. Tripakis. Counterexample classification. In R. Calinescu and C. S. Păsăreanu, editors, *Software Engineering and Formal Methods*, pages 312–331, Cham, 2021. Springer International Publishing.

[82] K. Viswanadha, E. Kim, F. Indaheng, D. J. Fremont, and S. A. Seshia. Parallel and multi-objective falsification with Scenic and VerifAI. In L. Feng and D. Fisman, editors, *Runtime Verification*, pages 265–276, Cham, 2021. Springer International Publishing.

[83] N. Volanschi and B. Serpette. Allenrv: An extensible monitor for multiple complex specifications with high reactivity. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification*, pages 393–401, Cham, 2019. Springer International Publishing.

[84] M. Yahyazadeh, S. R. Hussain, E. Hoque, and O. Chowdhury. PatrIoT: Policy assisted resilient programmable IoT system. In J. Deshmukh and D. Ničković, editors, *Runtime Verification*, pages 151–171, Cham, 2020. Springer International Publishing.

[85] E. Zapridou, E. Bartocci, and P. Katsaros. Runtime verification of autonomous driving systems in CARLA. In J. Deshmukh and D. Ničković, editors, *Runtime Verification*, pages 172–183, Cham, 2020. Springer International Publishing.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 8. Mai 2023