The Journal of Systems & Software 203 (2023) 111733

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

GRuM — A flexible model-driven runtime monitoring framework and its application to automated aerial and ground vehicles[☆]

Michael Vierhauser[a,*], Antonio Garmendia[b], Marco Stadler[a], Manuel Wimmer[c], Jane Cleland-Huang[d]

[a] *Johannes Kepler University Linz, LIT Secure and Correct Systems Lab, Linz, 4040, Austria*
[b] *Universidad Autónoma de Madrid, Departamento de Ingeniería Informática, Madrid, 28049, Spain*
[c] *Johannes Kepler University Linz, Institute for Business Informatics - Software Engineering & CDL-MINT, Linz, 4040, Austria*
[d] *University of Notre Dame, Department of Computer Science and Eng., Notre Dame, 46617, IN, United States*

ARTICLE INFO

Article history:
Received 23 October 2022
Received in revised form 22 February 2023
Accepted 25 April 2023
Available online 11 May 2023

Keywords:
Cyber-physical systems
Runtime monitoring
Model-driven engineering

ABSTRACT

Runtime monitoring is critical for ensuring safe operation and for enabling self-adaptive behavior of Cyber-Physical Systems (CPS). Monitors are established by identifying runtime properties of interest, creating probes to instrument the system, and defining constraints to be checked at runtime. For many systems, implementing and setting up a monitoring platform can be tedious and time-consuming, as generic monitoring platforms do not adequately cover domain-specific monitoring requirements. This situation is exacerbated when the System under Monitoring (*SuM*) evolves, requiring changes in the monitoring platform. Most existing approaches lack support for the automated generation and setup of monitors for diverse technologies and do not provide adequate support for dealing with system evolution. In this paper, we present *GRuM* (**G**enerating CPS **Ru**ntime **M**onitors), a framework that combines model-driven techniques and runtime monitoring, to automatically generate a customized monitoring platform for a given *SuM*. Relevant properties are captured in a Domain Model Fragment, and changes to the *SuM* can be easily accommodated by automatically regenerating the platform code. To demonstrate the feasibility and performance we evaluated *GRuM* against two different systems using TurtleBot robots and Unmanned Aerial Vehicles. Results show that *GRuM* facilitates the creation and evolution of a runtime monitoring platform with little effort and that the platform can handle a substantial amount of events and data.

1. Introduction

Robotic systems are increasingly used in the context of shop floor automation (Mayr-Dorn et al., 2021; Sykes and Keighren, 2018), as autonomous vehicles for medical device delivery (Eichleay et al., 2019), and search-and-rescue (Schörner et al., 2021) operations. Such Cyber-Physical Systems (CPS) exhibit tight integration between hardware and software components and frequently interact with humans. This in turn introduces a number of safety concerns that need to be mitigated (Nikolakis et al., 2019; Vierhauser et al., 2021b), resulting in the need for customized and system-specific runtime monitoring support. For example, when Unmanned Aerial Vehicles (UAVs) are engaged in search-and-rescue flights in close proximity to humans, or when robots operate on a factory floor, precautionary measures need to be taken to ensure that the CPS adheres to its specified requirements and operates within its predefined safety envelope. Therefore, support is required for monitoring diverse properties at runtime (Shakhatreh et al., 2019). However, the heterogeneity of hardware and software components within a CPS means that creating and implementing monitors, and subsequently collecting, processing, and checking the required data is often an arduous and time-consuming task.

Runtime information is typically collected from the System under Monitoring (*SuM*), through source code instrumentation (Li et al., 2013; Eichelberger and Schmid, 2014a), using dedicated data-buses (Hamida et al., 2012; Popescu et al., 2010), or other collection services. It is then processed at runtime to check constraints to ascertain whether the system is behaving according to its specified requirements, or if deviations from expected behavior have occurred. While off-the-shelf monitoring approaches support application performance monitoring (Eichelberger and Schmid, 2014a) or checking Service Level Agreements (Ruz et al.,

☆ Editor: P. Lago.
* Corresponding author.
E-mail addresses: michael.vierhauser@jku.at (M. Vierhauser), antonio.garmendia@jku.at (A. Garmendia), marco.stadler@jku.at (M. Stadler), manuel.wimmer@jku.at (M. Wimmer), janeclelandhuang@nd.edu (J. Cleland-Huang).

2010; Keller and Ludwig, 2003), they often provide inadequate support for instrumenting custom systems, or defining complex temporal or structural constraints. The problem is exacerbated as CPS are typically long-running, with ongoing maintenance and evolution activities that require both, the data collection mechanisms and constraints to be updated and adapted. If the monitoring infrastructure does not co-evolve with the *SuM*, constraints may become stale, and data outdated due to changes in the underlying components. Few approaches have addressed the challenge of automatically generating customized, system-specific, runtime monitoring solutions and their maintenance and evolution support. In closely related work, Model-Driven Engineering (MDE), which automatically generates source code from system models, has previously been used to model entire systems, including their runtime properties (Reynolds et al., 2020; Brand and Giese, 2019). When a new feature or functionality is introduced, the model is updated, and its respective code regenerated. As a result, MDE has been shown to improve productivity by enabling developers to specify a system at a higher level of abstraction. However, the application of MDE has fallen short of enabling the generation and evolution of a complete monitoring platform (Rabiser et al., 2017).

To address these shortcomings, we propose and evaluate *GRuM*, a model-driven framework for **G**enerating CPS **Ru**ntime **M**onitors. We aim to enable (i) the generation of a customized monitoring platform with support for data collection and analysis, which (ii) can be readily extended and updated when changes to the monitored system occur. *GRuM* provides a lightweight monitoring solution that only requires modeling those parts of the system that are relevant for the monitoring infrastructure. As a result, it can be applied to diverse projects, whether or not MDE is adopted as the primary development paradigm. This paper builds upon our earlier work (Vierhauser et al., 2021a; Stadler et al., 2022), in which we collected requirements for model-driven monitoring and derived an initial architecture. We significantly extend that work by introducing a refined architecture, extending our weaving and meta-model, and providing code generators for the monitoring platform. Furthermore, we have greatly extended the automation support for automatically generating the platform and monitors, and include a thorough evaluation of *GRuM* using two diverse systems. More specifically, we evaluate our approach against a system for deploying UAVs for search-and-rescue and a second system that uses a set of mobile robots to assess indoor air quality.

The contributions of this work are as follows: First, we present a monitoring meta-model that enables us to generate a custom monitoring platform with a runtime model, instead of implementing it manually. This provides a higher automation potential, for more efficient development of monitoring solutions. Second, probes are automatically generated for the target technology. Given this platform, we can leverage different types of (off-the-shelf) constraint engines depending on the monitoring needs, as shown in our evaluation (using a CEP Engine for temporal constraints and event sequences) without the need to reimplement other parts. Finally, once a probe generator for a specific technology has been implemented, evolving the monitoring framework alongside the *SuM* is greatly simplified and only requires a few steps to update the model and automatically regenerate the platform.[1]

The remainder of the paper is structured as follows: Section 2 describes open challenges. Section 3 presents an overview of the *GRuM* framework, while Section 4 describes its application and implementation. In Section 5 we first describe our evaluation objectives and setup, and in Sections 6 to 8 we report on our evaluation results for two real-world systems and discuss results and threats to validity. Finally, Section 9 describes related work and, Section 10 draws conclusions and proposes future directions.

2. Monitoring challenges and motivating example

Runtime monitoring plays a pivotal role in checking and ensuring that a system operates safely within its predefined operational capabilities and that it adheres to its specified constraints related to both functional and non-functional requirements related to performance, safety, and other quality concerns. It plays an intrinsic role in creating Digital Shadows and Digital Twins (Kritzinger et al., 2018; Uhlemann et al., 2017), and provides support for the MAPE-K loop in self-adaptive systems (Weyns et al., 2013; Elgendi et al., 2019).

However, the field of runtime monitoring is quite diverse with approaches that have been designed to support various purposes, technologies, and types of system architectures (Vierhauser et al., 2016a; Zavala et al., 2019). Examples include service-based systems (Keller and Ludwig, 2003; Baresi and Guinea, 2005), application performance monitoring (Eichelberger and Schmid, 2014a), and goal-oriented monitoring approaches (Wang et al., 2009; Baresi et al., 2010). The fact that existing approaches are typically customized for a specific type of system, application domain, or support particular types of constraints, has further hampered the broader adoption of runtime monitoring frameworks. As a result, significant upfront investment is required to implement and maintain custom monitoring infrastructures, often without inbuilt support for maintenance and evolution after deployment. In previous work, Rabiser et al. (2019) investigated several different monitoring frameworks and ultimately derived a reference architecture, identifying three common components: *Monitoring Setup* related to defining monitors, generating probes, and instrumenting a system, *Monitoring Execution* concerning data collection, processing and constraint checking, and finally *Monitoring Support* targeting additional functionality such as data persistence and external applications.

The continuous maintenance, enhancement, and hence evolution, that complex CPS are subject to, has a significant impact on any monitoring infrastructure that collects and analyses data from a system. Runtime monitoring frameworks need to embrace these changes and co-evolve with the *SuM*, providing support for updating, and maintaining the monitoring framework itself as part of the system development process (Martínez-Fernández et al., 2019; France and Rumpe, 2007).

These challenges are illustrated in *Dronology*, an open-source, multi-UAV system (Cleland-Huang et al., 2018) with support for planning and executing UAV missions. Establishing a monitor for Dronology requires in-depth knowledge of the system in order to identify and collect necessary data, establish a monitoring platform that aggregates data from different UAVs and then stores the data for subsequent analysis, and ultimately to select and configure a suitable constraint engine to analyze the data and provide evaluation results. In the case of Dronology, instrumentation is needed to collect UAV data (e.g., GPS coordinates, attitude, mission status), transmit the data to a Ground Control Station which is responsible for aggregating, analyzing, and persisting the data, and developing a dashboard for visualizing data to the UAV operators. In addition, a constraint engine needs to be configured to check constraints, such as whether the UAVs remain within their designated altitude band, avoid no-fly zones, and are flight-worthy.

Ongoing changes to both, the hardware and software can adversely affect the monitoring system. For example, if a new

[1] All models and source code, as well as the generated monitoring infrastructures, are available on GitHub as an open-source system at https://github.com/LIT-Rumors/grum-public.

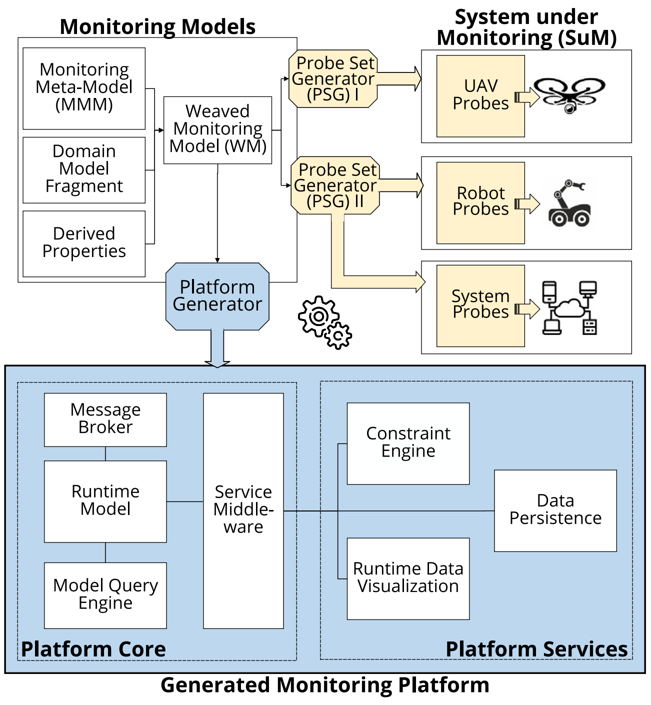


Fig. 1. Architectural overview of the *GRuM* components for modeling the domain and relevant constraints, and for generating probes and the core monitoring platform.

water sampling capability were introduced to Dronology, it would require additions and modifications to the hardware and software components. New downward-facing sensors would be required to ensure that the UAV maintains a stable distance from the water during the collection process. Furthermore, as light reflecting from the water could cause dangerous altitude fluctuations, the monitoring system would need to be updated to continually check that altitude remained within an acceptable range. In order to reduce the cost and effort of creating and maintaining a runtime monitor as illustrated for Dronology, it is necessary to (*i*) provide automated support for creating a system-specific solution, including the setup and configuration of the monitoring system and specification of its respective constraints, and (*ii*) avoid the additional effort of maintaining the monitoring system itself by enabling the infrastructure to co-evolve alongside the *SuM*.

3. The *GRuM* framework

To provide support for the different parts of a runtime monitoring architecture and to ease the task of maintaining and co-evolving monitors, *GRuM* leverages MDE techniques to specify relevant monitoring properties and ultimately to generate a complete monitoring infrastructure. It follows Rabiser et al.'s reference architecture (Rabiser et al., 2019), providing support for data collection, analysis (i.e., checking constraints), and visualization. However, one of the novel characteristics of *GRuM* is that for a *SuM*, both a *Set of Probes*, as well as a fully customized *Monitoring Platform* can be generated based on a model describing the parts to be monitored. Fig. 1 provides an overview of *GRuM*'s architecture. The *Modeling* part relies on a Monitoring Meta-Model (MMM) and a system-specific Domain Model Fragment that is populated for a *SuM*; the *Code Generators* responsible for generating the Monitoring Platform and the probes for collecting data from the *SuM*; and finally the generated *Monitoring Platform* itself, providing a number of runtime capabilities such as a runtime model, a query engine, and a middleware layer to connect external services. Properties of interest from the *SuM* that are collected and monitored via Probes can be accessed in various different ways. This, to a large extent depends on the code generator, and the resulting Probes that are generated from the model. For example, when generating a simple monitoring API (cf. Section 4), the *SuM* needs to be "accessible", meaning that API calls can be added. If additional means of data collection are required, a code generation that used byte-code instrumentation (Vierhauser et al., 2016a; Goldberg and Haveland, 2003), or aspect orientation (Cassar et al., 2017) might be employed.

3.1. *Monitoring models*

MDE has been used extensively to generate applications and system components across a wide variety of domains, including automotive, railroad systems, business process engineering, and embedded systems (Bucchiarone et al., 2020; Quiñones et al., 2020; Asici et al., 2019). *GRuM* uses model-driven techniques to generate a customized monitoring platform for the *SuM*. It only requires the parts of the *SuM* that need monitoring to be modeled, instead of the entire *SuM*. This represents a significant time-saving as it is often the case that only a small subset of properties and data created by a system are of interest at runtime. For example, a UAV's PX4 flight controller (PX4, 2021) has over 1200 configurable properties and sensor values. Of these, a few values are regularly checked during the preflight arming process, whilst other values, such as GPS coordinates, altitude, attitude, velocity, satellite links, temperature, and camera/gimbal settings are typically monitored during flight; however, many hundreds of properties, such as internal flight controller states are unlikely to be monitored at runtime. The *GRuM* Monitoring Setup specifies exactly what parts of the system should be monitored and provides the mechanisms for performing the monitoring.

The challenge of creating a *Probe* (Mansouri-Samani and Sloman, 1993), to collect runtime information from the *SuM* and making it available to the monitoring infrastructure, is that each *SuM* uses different technologies, diverse architectural styles, and consists of various software and hardware components. Therefore, in practice, monitors are typically defined and developed individually for each type of system or technology, such as byte-code instrumentation or dedicated service busses in service-based systems (Eichelberger and Schmid, 2014a; Popescu et al., 2010).

GRuM addresses this challenge by providing a generic approach for describing monitoring properties and their resulting monitors, so that instead of building a custom set of probes, we specify the setup in a standardized way and generate a custom monitor directly from the specification. *GRuM* leverages MDE to describe the components of the *SuM* that need to be monitored, and then subsequently generates the monitoring infrastructure. To support this process we have created a dedicated MMM that can be used to define the relevant parts of the system for the resulting probes and the Monitoring Platform.

3.1.1. *Monitoring Meta-Model (MMM)*

Our MMM, shown in Fig. 2 was derived from analyzing the monitoring reference architecture from Rabiser et al. and existing runtime monitoring frameworks (Rabiser et al., 2019). The `MoConfig` element defines the monitoring configuration for a specific system and is subsequently used to generate the code for a target system (cf. Section 3.2). It contains information regarding the monitoring setup, e.g., the server configuration (`Server`), and it specifies elements of the system to be monitored. Monitorable agents (`MoAgent`) represent elements of the system, such as individual machines, devices, or robots that are

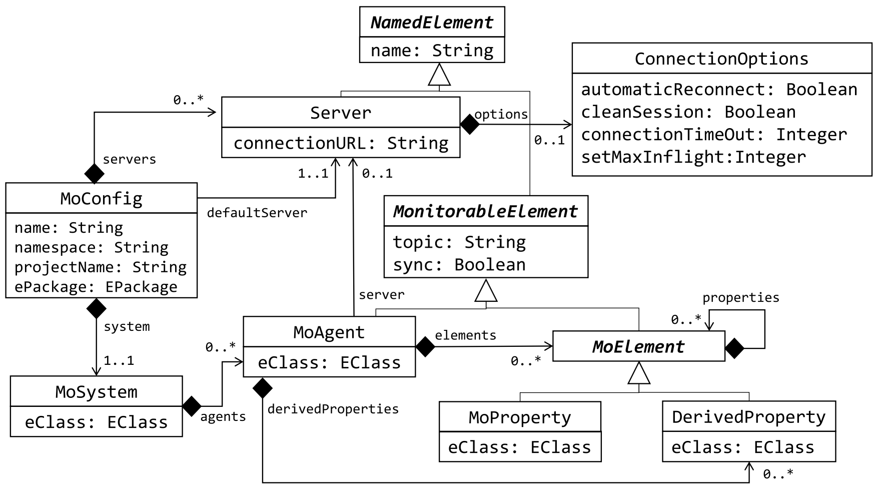


Fig. 2. *GRuM*'s Monitoring Meta-model (MMM).

subjects for monitoring. Each agent has its own set of monitorable properties (MoProperty) that are exposed to the monitoring framework and used for checking constraints or logging. Finally, the MMM provides support for defining derived properties (DerivedProperty) that are a result of a post-processing analysis such as data aggregation or querying the data model (cf. Section 3.2.2).

3.1.2. Domain model fragment

The MMM provides the foundation for all systems monitored with *GRuM*, requiring only those properties of the *SuM* that must be collected and analyzed at runtime to be specified in a *Domain Model Fragment*. Fig. 3 contains a partial Domain Model Fragment for the Dronology system, showing the Drone elements, each of which consists of different properties, such as the state (DroneState), a dedicated flightplan (FlightPlan), and safety checks at startup (StartUpCheck). The state, in turn, includes sub-properties such as battery and location information. While a Drone has many other properties that are either set during startup or at runtime, the Domain Model Fragment describes the subset of properties to be collected and analyzed by the monitoring framework. Properties can represent primitive data types (e.g., numbers, strings) or more complex aggregated elements.

3.1.3. Weaved monitoring model

In order to support the generation of monitoring code, the different elements specified in the Domain Model Fragment need to be linked to the concepts specified on the MMM.

These links are specified in the *Weaved Monitoring Model (WM)* and are subsequently used to determine which code fragments are generated for which element in the Domain Model Fragment. Model weaving has been used in the past to connect different models to establish dedicated links between them (Del Fabro et al., 2005a,b, 2006; Jouault et al., 2010; Hawkins et al., 2015). As described by Del Fabro et al. (2006), the weaving model is not self-contained but is meant to be used in conjunction with its related models and to provide the respective link semantics for the application scenario—in our case, the creation of the runtime monitoring platform.

In the WM, for example, a Drone element from the Dronology Domain Model Fragment is linked to the *MonitorableAgent* from the MMM, whilst DroneState and FlightPlan are children of the Drone element and linked to a *MonitorableProperty* as they can be directly collected from the system. The resulting WM (cf. Listing 1) is a hierarchical representation of the monitored system. Additionally, the WM also contains information about the topic mappings as well as the server configuration (represented by the MoConfig element) which is used to automatically generate topic subscriptions and connection code.

This clear separation of concerns, the monitoring information on the one hand, and the system configuration on the other, facilitates easy updates of the Domain Model Fragment when new properties or agents are added. It also supports reuse of existing models in cases where new *SuMs* also use model-driven techniques to generate system code.

3.2. Code generation

A runtime monitoring platform relies upon several different components to collect, aggregate, and analyze runtime data. Based on the information encoded in the WM and the Domain Model Fragment, *GRuM* automatically generates (i) a set of probes for collecting information from the system, and (ii) an instance of the Monitoring Platform for the *SuM*.

```
MoConfig DroneConf
   projectName DroneProject
   defaultServer localserver
   ePackage Dronology
   servers{
      Server localServer
      ConnectionURL "tcp://192.168.0.55"
   }
   system MoSystem {
       agents {
         MoAgent "Drone" {
         eClass "Dronology.Drone"
           elements {
             MoProperty "DroneState"
             topic "state"
             sync true
             eClass "Dronology.Dronestate"
  ... }
```

Listing 1: Excerpt of the WM for the Dronology system.

3.2.1. SuM probes

Existing monitoring approaches use a variety of techniques to collect information from the running system. In order to reduce the overhead of manually implementing system-specific probes and to create data collection mechanisms for each system individually, *GRuM* generates a technology and language-specific set of probes. The Probe Set Generator (PSG) is independent of the data that is actually collected from the system but is dependent upon the underlying technology. For example, a PSG for a Java-based system that generates target Java code can be reused for

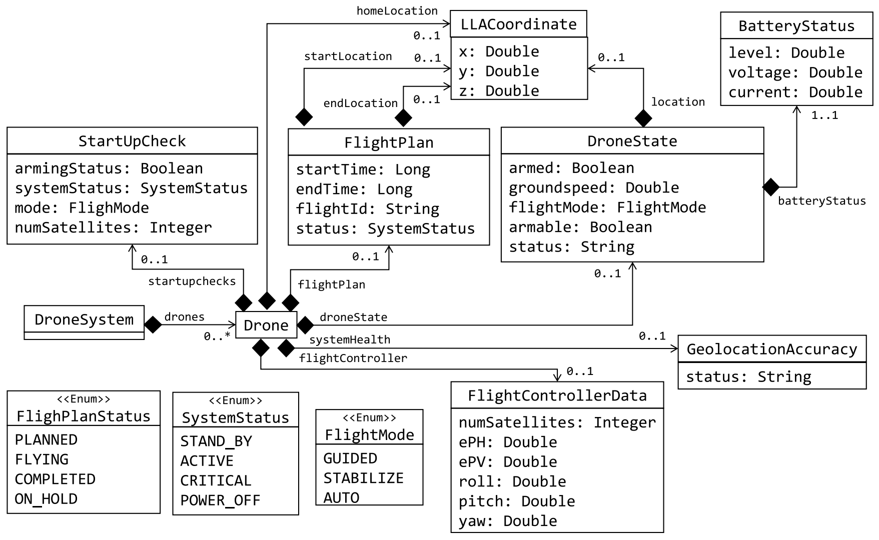


Fig. 3. Excerpt of Dronology's Domain Model Fragment.

other Java-based systems, and code can be re-generated when new properties or types of agents are added to the Domain Model Fragment. New PSGs can be easily added to *GRuM* when probes for a new technology or type of system are required without any need to redesign or re-implement the underlying monitoring platform.

3.2.2. Monitoring platform

Runtime data provided via probes subsequently has to be collected and aggregated, so that the data can be analyzed, constraints checked, and results visualized. A topic-based message broker sends runtime data from the *SuM* (via the probes) to the generated infrastructure. The respective topics and topic subscriptions are generated automatically based on the agents and properties defined in the WM. Each `MoAgent` represents a root topic (e.g., "Drone"), and each `MoProperty` a respective subtopic, e.g., "State". At runtime, each activated drone sends information through these topics (e.g., "Drone/Drone1/State" for an update of the State property). This data is then used to populate and update the runtime model.

Every change in the model triggers a query against the *Model Query Engine* where constraints, i.e., checks on the model can be defined. Constraints are then evaluated on the runtime model and violations are generated when an evaluation fails. For example, for the drone system, a constraint in the query engine could check whether the reported FlightPlan received from the probe contains a valid flight id and a set of valid coordinates the UAV should fly to. In case the constraint check fails, a violation is generated which can then be displayed in the user interface and the user can cancel the flight, or if desired, the flight could be aborted automatically.

3.2.3. Monitoring middleware

In addition to the monitoring platform, *GRuM* also generates a middleware component that provides a *SuM*-specific interface to, for example, connect external services or applications. This allows applications to register to specific properties, and receive notifications when a change in the model occurs, for example, when a new agent has been added, or when properties are updated. Each agent and property is thereby accessible via a dedicated API method that corresponds to information encoded in the WM (Listing 1). The mapping between the updated property/agent and the topic where the information is published in the middleware is established via the topic name specified in the WM. Examples of external applications that can be attached include additional constraint engines to support specific types of constraints (e.g., temporal constraints or event patterns), attaching a database to store runtime data, or adding a new UI. Separating the platform's core capabilities from external services decouples the data-receiving parts of the *SuM* from application services that build on top of it. The interface provides access to all information collected and aggregated in the runtime model from the *SuM*.

4. Applying *GRuM*

The process of using *GRuM* to specify and generate a *monitoring platform* and then deploying the platform to monitor a system at runtime is described in Fig. 4. The upper part involves creating the respective models and generating the platform code, whilst the lower part uses the resulting platform to define constraints and to connect external services, before deploying the platform for the *SuM*.

4.1. Creating the models

In the first step of the process, relevant system information needs to be identified. This could include hardware and software components, such as sensors and their respective data, that provide valuable or critical runtime information. Various approaches can be used to identify and document relevant runtime requirements, safety properties, or QoS attributes (Rabiser et al., 2017; Aizawa et al., 2018; Maiden, 2013; Caldiera and Rombach, 1994; Galster and Bucherer, 2008); however, the identified properties must be (1) specified in the *SuM*'s *Domain Model Fragment*. In case the *SuM* itself is model-driven, existing models of the system can be leveraged. Once the Domain Model Fragment is populated with an initial set of properties and attributes, these are (2) linked to the different monitoring concepts via the *Weaved Monitoring Model*. The WM captures monitored data, structured by the different agents and their constituent monitorable properties. This step is vital for the subsequent code generation, so that the platform code generator can generate the specific runtime model and the hierarchical topic structure.

4.2. Platform and probe generation

Based on the WM, (3) two different types of monitoring code is generated for the *SuM*. First, a set of probes is generated using a

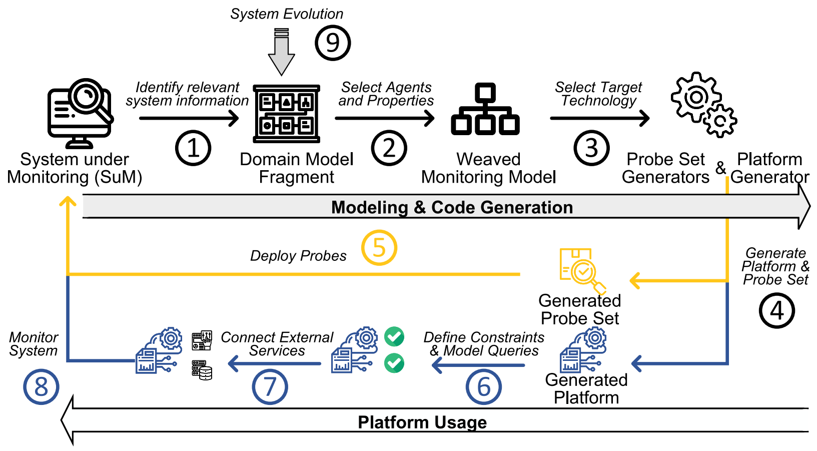

System Evolution
9
Identify relevant system information
1
Select Agents and Properties
2
Select Target Technology
3
System under Monitoring (SuM)
Domain Model Fragment
Weaved Monitoring Model
Probe Set Generators & Platform Generator
Modeling & Code Generation
Deploy Probes
5
Generate Platform & Probe Set
4
Generated Probe Set
Monitor System
8
Connect External Services
7
Define Constraints & Model Queries
6
Generated Platform
Platform Usage

Fig. 4. *GRuM* process for specifying, generating, and utilizing a monitoring platform.

technology-specific PSG which is either selected from an existing library or created from scratch. *GRuM* provides templates to ease the implementation of these generators, and new PSGs can be stored in the library and then reused for any other *SuM* using the same technology. The second code generator is technology-independent and is shared across all *SuM* applications. It (4) generates the monitoring platform containing the runtime model, data aggregation, and analysis component.

4.3. Platform usage & runtime monitoring

Generated probes can be (5) directly deployed to the *SuM*. Depending upon their type (e.g., byte-code instrumentation probes, or data interceptors), they are integrated directly by the developers, or as part of the continuous integration process, into the source code or the binary files of the system. The generated platform then allows (6) users to define constraints and runtime checks. *GRuM* deliberately does not store constraint information in the Domain Model Fragment, and hence does not generate the constraints, since these are typically added incrementally, and subject to modifications at runtime (Zavala et al., 2019). *GRuM* supports constraint checks in two different ways. First, it provides an integrated model query engine that can query `MoProperties` and second, it is used to define checks on the model, for example, to ensure that a property stays within a certain range. The monitoring middleware generates notifications when a constraint is violated. Additional constraint engines can be connected via the middleware, providing the connectivity needed for constraint violations to be detected and reported. All additional services (7) are connected via the middleware layer, and once connected, the platform (8) can be deployed to collect and analyze runtime data sent from *SuM* via the generated probes.

4.4. System evolution

Since all major components of the platform and the probes are generated by *GRuM*, changes made to the *SuM* can be easily incorporated into the monitoring platform as well. This significantly reduces the effort of co-evolving the monitoring platform, as well as the probes with the *SuM*. Changes in the *SuM* can be reflected in the Domain Model Fragment and WM, and the code for the probes and platform can then be automatically re-generated. Previously defined constraints as well as external services remain unaffected and can be reused or deactivated (in case a constraint is checking a specific property that no longer exists, or is not monitored anymore). For example, if an additional property of an existing agent needs to be added to the set of monitored properties, this change can be directly performed at the model level. The respective element is first added to the Domain Model Fragment and then linked in the WM. Once this change has been made, probes and platform code can be completely regenerated without the need to manually adapt the code. In case a property is removed for which a model query has been defined the query engine provides an error message in the query description so that stale and invalid constraints can be ignored or removed.

4.5. GRuM implementation

We implemented a fully operational prototype[2] of *GRuM* supporting the aforementioned parts based on the Eclipse Modeling Framework (EMF) (Foundaition, 2021). EMF provides capabilities to describe meta-models by using its Ecore language which we used for the MMM and the Domain Model Fragment. For the automatic code generation, we use Roaster (JBoss, 2021) to parse and create Java source files. Additionally, we used XText (Eclipse Foundation, 2021) to define a textual domain-specific language (DSL) for the WM (see example in Listing 1) which allows easy creation and modification of the WM for a specific *SuM*.

Probe Set Generator: As part of *GRuM* we provide a template and utility functions (e.g., for packaging and deployment) to create an implementation of a PSG for a target language/technology. A concrete implementation needs to iterate over the elements defined in the WM and then generate topic mappings for the respective agents and their constituent properties, as well as the code to send the data to the message broker. As part of our evaluation, we have implemented two distinct code generators to support two different target languages and technologies. The first generates a Java-based Probe Set, whilst the second targets ROS-based systems and generates a Python component with ROS-node subscriptions (cf. Section 6).

Monitoring Platform Generator: As EMF natively generates Java code from Ecore models, we leverage this feature, and our runtime model and the platform core components are implemented in Java (i.e., Java code is generated by the platform generator for each *SuM*) regardless of the *SuM*'s native programming language. At runtime, an instance of the Domain Model Fragment is created and used as the runtime model. All model code, code for instantiating the model, as glue code for setting up the platform is generated automatically. For data transmission, we use MQTT

[2] Further implementation details and documentation is provided in our GitHub repository: https://github.com/LIT-Rumors/grum-public.

(2023), which provides a scalable, topic-based publish and subscribe mechanism which is also automatically configured based on the WM.

The generated middleware layer is also Java-based and for each property and agent specified in the model, a respective notification method is generated. This allows external applications to register to be notified about changes in the runtime model. Furthermore, data is published on predefined topics (as specified in the WM), which can directly be accessed by any service or application subscribed to the respective topic via the MQTT broker. In case multiple instances of an agent are active (e.g., three UAVs are monitored at the same time) the messages are augmented with the respective "agent id" so that property updates received via the middleware can be assigned to the proper agent. This separation between the incoming information from the probes and outgoing information to the external services ensures that all information is handled by the runtime model which serves as the single point of information for all connected services.

Constraint Evaluation & Visualization: Integrated within the monitoring platform core, our implementation uses the Viatra model query engine, which is tightly integrated into EMF and directly operates on the model for specifying checks on runtime properties. However, while Viatra provides tools and incremental evaluation (Bergmann et al., 2010, 2015), it does not support more complex constraints, such as temporal ones. We, therefore, connected the Esper CEP engine (Espertech, 2021) via the middleware to support Complex Event Processing (CEP) as an additional service for checking temporal sequences and occurrences of events.

5. Evaluation objectives and setup

When evaluating our *GRuM* framework we address three main aspects. First, we address the general applicability of our approach to diverse systems that should be monitored. Second, we focus on the evolution aspect where we assess the effort that is in fact required when the *SuM* evolves compared to other monitoring approaches. Finally, we target efficiency and evaluate the performance of the generated monitoring platform to ensure its suitability in real-world application scenarios. For this purpose we used our *GRuM* implementation and investigate the following research questions:

RQ1: Can GRuM be used to create a runtime monitoring platform for a CPS with reasonable effort?

While representing a binary question, it is an important one for validating that *GRuM* works as specified. We, therefore, address the question by using *GRuM* to generate a monitoring infrastructure, collect events from the system and evaluate constraints at runtime.

RQ2: What additional effort is required to update the platform and monitors when the SuM evolves?

With the second research question we start with the generated platform (RQ1) and then analyze the effort needed to co-evolve *GRuM*'s models and monitoring platform when the *SuM* is modified. The goal hereby is to assess how different change scenarios impact *GRuM* and the resulting changes required.

RQ3: To what extent can GRuM's generated code be used to efficiently monitor time-sensitive runtime data in a CPS system?

The last RQ directly addresses the pertinent question of whether the code generated by *GRuM* is efficient. This is an important question, as generated generic code is often perceived to be less efficient than manually constructed code, and efficiency is particularly important in a runtime monitoring system. We address this question by measuring the model-set-latency of the monitoring platform, i.e., by measuring the time it takes to update the model at runtime when different property values change.

Two different researchers, both co-authors of this paper, performed the tasks associated with research questions RQ1 and RQ2. One was assigned to Dronology and one to the TurtleBot system. Both researchers were familiar with the respective *SuM*, had previous experience in Java development (for the Dronology use case) and ROS and Python (for the ROS TurtleBot use case), as well as basic experience with Eclipse and the Eclipse Modeling Framework for creating Ecore models. Specific knowledge about code generation with Xtend was only necessary for implementing the initial Probe Generator prototypes, but not for subsequently generating a new *GRuM* instance based on the existing generators.

5.1. Use cases

We explored the three research questions in two diverse robotic systems.

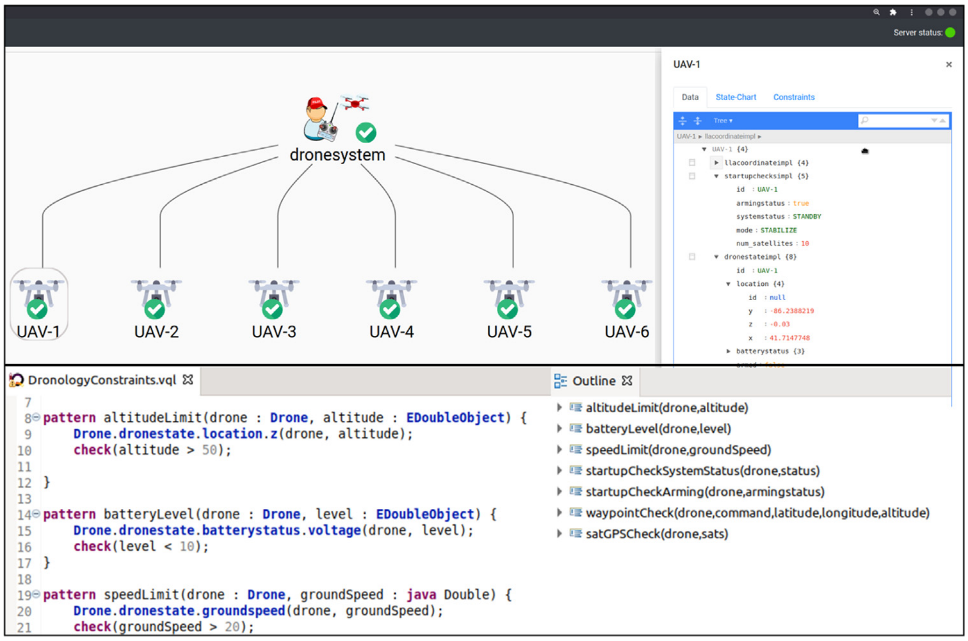
Case 1 – Dronology: The first use case was the previously discussed Dronology UAV system which we developed from 2016 to 2020 using Java and DroneKit (Cleland-Huang et al., 2018) and is now available in the open domain. Dronology is fully compatible with both physical UAVs and simulated ones and has been used in extensive field deployments. The only difference between simulations and physical deployments is a single string used to establish communication with a real or simulated UAV. For our experiments, we utilized the high-fidelity Ardupilot simulation. All parts of the Dronology architecture pertaining to UAV management (such as route creation, mission planning, user interfaces, etc.,) are Java-based, while only the Groundcontrol Station responsible for forwarding messages to the UAV's flight controllers is Python-based. In our evaluation we, therefore, focus on the Java system and generate the Java monitoring Probes for relevant parts.[3]

Case 2 – ROS TurtleBot Robots The second system used TurtleBot3 robots (Robotis, 2021) equipped with sensors to collect $C0_2$ measurements (cf. Fig. 6). A TurtleBot is a small mobile robot using the Robot Operating System (ROS) (Quigley et al., 2009) as a platform which has emerged as a de-facto standard for both research and industry. The TurtleBot platform is also used frequently for prototyping in research and education in the area of CPS in general and robotics in particular (Aagela et al., 2017; Koubâa et al., 2016; Amsters and Slaets, 2019). ROS is an open-source platform for robotics software development using a component-based architecture with nodes that interact via a publish–subscribe pattern. ROS supports both hardware and high-fidelity simulation using, for example, the Gazebo simulation environment (Open Robotics, 2023). As ROS applications are commonly implemented in Python, we used this second case to demonstrate the applicability of *GRuM* with different target languages and technologies. As part of the prototype we implemented an application for controlling the TurtleBots and a number of control scripts for performing actions. Additionally, to augment the original Hardware of the TurtleBot3 robot, we attached an MQ-135 $C0_2$ sensor to facilitate mobile sensor collection with the robots. All required parts for this case were implemented in Python, and further details can be found in the open-source GitHub repository.

6. Evaluation

In the following we address our three research questions for creating (RQ1), evolving (RQ2) and evaluating the performance (RQ3) of *GRuM*.

[3] Furhter details about the architecture, requirements, etc. can also be found on the project website: https://dronology.info.



```
▾ UAV-1 {4}
    ▸ llacoordinateimpl {4}
    ▾ startupchecksimpl {5}
        id  : UAV-1
        armingstatus : true
        systemstatus : STANDBY
        mode : STABILIZE
        num_satellites : 10
    ▾ dronestateimpl {8}
        id  : UAV-1
        ▾ location {4}
            id  : null
            y   : -86.2388219
            z   : -0.03
            x   : 41.7147748
        ▸ batterystatus {3}
```

```

pattern altitudeLimit(drone : Drone, altitude : EDoubleObject) {
    Drone.dronestate.location.z(drone, altitude);
    check(altitude > 50);

}

pattern batteryLevel(drone : Drone, level : EDoubleObject) {
    Drone.dronestate.batterystatus.voltage(drone, level);
    check(level < 10);
}

pattern speedLimit(drone : Drone, groundSpeed : java Double) {
    Drone.dronestate.groundspeed(drone, groundSpeed);
    check(groundSpeed > 20);
```

```
▸ altitudeLimit(drone,altitude)
▸ batteryLevel(drone,level)
▸ speedLimit(drone,groundSpeed)
▸ startupCheckSystemStatus(drone,status)
▸ startupCheckArming(drone,armingstatus)
▸ waypointCheck(drone,command,latitude,longitude,altitude)
▸ satGPSCheck(drone,sats)
```

Fig. 5. User Interface Prototype (upper half) and examples of Viatra model queries (lower half).

6.1. RQ1 – Creating a GRuM instance

In the first part of our evaluation, we used *GRuM* to model MoAgents and MoProperties, created the WM, implemented the reusable PSGs needed for each system, generated the probes and monitoring platform for each system, and specified constraints against the runtime model. Two different researchers, both co-authors of this paper, performed this task. One was assigned to Dronology and one to the TurtleBot system. We then recorded the time taken to complete each activity.

Case 1 – Dronology: As we had not previously created a PSG for supporting probes in Java systems, we started by creating such a generator. Based on *GRuM*'s template for generating deployment packages we used a Java code generation framework (JBoss Roaster) to specify source code for the Java probes to be generated. We implemented a push-based probe generator that used the Domain Model Fragment and the WM (parts shown in Listing 1) to generate dedicated publish methods for each property, as well as utility classes for setting up the connection to the MQTT broker. The implementation effort for creating this reusable code generator was approximately 10 h.

Modeling & Code Generation: We then developed runtime monitors for the Dronology system. This was guided by our inherent knowledge of the UAV domain and the Dronology system, including its properties, and the constraints that should be checked at runtime. From this, we (1) specified an initial Domain Model Fragment using the Eclipse EMF Ecore Editor. The model included multiple UAVs, representing monitorable agents, controlled by the `DroneSystem`, and several drone-related properties, such as `DroneState` and, `DroneCommands`. In total, we modeled 8 properties with 33 attributes. We also modeled `GeolocationAccuracy` as a derived property computed from each UAV's GPS status and its reported bias. We then (2) created the WM specifying the server configuration and the hierarchical structure of the system. These two steps took around 1–2 h of effort. As the process was informed by our in-depth knowledge of the system, the reported effort focuses purely on the task of establishing the monitors rather than on "thinking-time" which is expected to be similar whether the monitor is built from scratch or using *GRuM*. We then (3) selected the Java code generator that we had created and

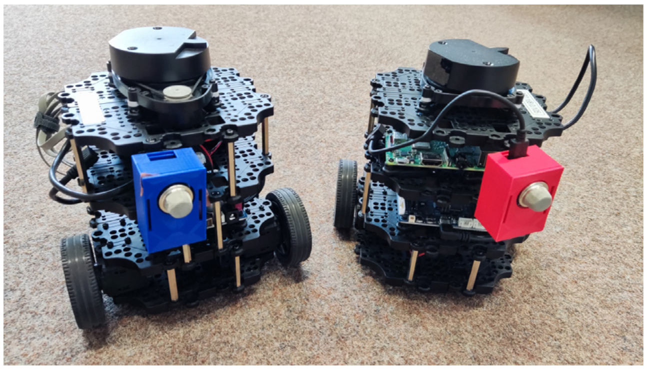


Fig. 6. Two of the TurtleBot robots equipped with an additional CO_2 sensor.

(4) generated an initial version of the Dronology monitoring platform and its associated probes, composed of a set of executable Eclipse Plugins (for the platform) and a maven project (for the probe) that were directly integrated into Dronology (5). The only modifications made directly to the Dronology code base were the insertion of one method call to the probes for each monitorable property. All other functionality that was needed to collect data, establish a connection to the platform, and publish data, was fully covered by generated code.

Platform Usage: We then derived a set of constraints (6). Once again, these were guided by our own knowledge of the Dronology system (Software, 2021) and its deployment in real-world tests. In total, we created 10 constraints, 7 of which were implemented using the Viatra model query engine (Bergmann et al., 2015), with three additional temporal constraints implemented using the Esper CEP Engine (Espertech, 2021), as listed in Table 3. Additionally, we (7) developed a generic UI for visualizing agents and runtime data (cf. Fig. 5). Both the Esper CEP engine and our UI were connected via the middleware. Finally, we (8) deployed the platform for the subsequent evaluation. Altogether, the implementation of constraints and extensions was completed within about 20 h.

Case 2 – ROS TurtleBot: As with the Dronology example, we started by creating a new PSG. For the TurtleBots, the generated code targeted Python to collect data from appropriate ROS topics and forward it to our monitoring platform. We modeled it in the same way as the Java-based generator by implementing a push-based approach. The Probes were implemented as a custom ROS package, consisting of the *ROS API* which provides update-functions for every monitored property of the ROS system, and an *MQTT-Forwarder* for sending data to the monitoring platform. Generated probes were implemented using ROS Noetic Ninjemys distribution packages in combination with the rospy client library for ROS.

Modeling & Code Generation: In contrast to the Dronology system, we developed this application from scratch without prior experience with the system or technology. Therefore, as a starting point, one researcher reviewed documentation for the TurtleBot platform (Robotis, 2021), and identified relevant properties and their associated ROS topics over which the data was published. The resulting Domain Model Fragment (1) for the TurtleBot consisted of 10 properties with 23 attributes. We then (2) again created the WM for the Bot system specifying the server configuration and the hierarchical structure of the system. It took approximately 15 h to investigate ROS properties, identify relevant ROS messages and create the Domain Model Fragment and WM. Of this, we estimate that 5 h were spent implementing the *GRuM* monitor after properties had been selected.

Platform Usage: After (4) selecting the Python probe generator, the resulting probes were (5) easily integrated into the ROS workspace and executed without further customization. For the constraints (6) we again created 7 constraints for the Viatra query engine and 3 additional temporal constraints with Esper. In this case, constraints included a maximum carbon dioxide concentration and an enforced speed limit when low battery is detected to lengthen battery life and allow the bot to return to its starting position. A complete list of constraints is provided in Table 3. We (7) reused the same generic user interface as developed for the Dronology system, and (8) deployed the platform. In order to evaluate whether *GRuM* can easily cope with changes in the *SuM*, (9) we additionally mounted an external air quality sensor (MQ 135) and connected it to the control board of the TurtleBot. We incorporated this change in the Domain Model Fragment by adding an additional monitorable property and then regenerated the probes and the monitoring platform. The two constraints CST-T06 and CST-T10 specifically use the data reported by the $C0_2$ sensor to check that the value does not exceed a critical threshold and that measurements are collected only when the robot is stationary.

Analysis of Implementation Process: Creating the Domain Model Fragment in both cases was a straightforward task. After analyzing the two systems and selecting properties to be monitored, EMF provided an easy way to create the model, and was similar to using any other UML modeling tool. The implementation of each PSG required an initial one-time effort but the PSG was then reused to support subsequent changes in the system and Domain Model Fragment. Furthermore, once a generator for a certain language and/or technology is made available, it can be reused for any other system (in our case for any other Java-based or ROS-based Python application). To answer RQ1, for both cases we were able to model the monitoring properties with minimal effort. The vast majority of this time was used to specify (and test) temporal constraints with Esper, as no immediate tool support was provided, in contrast to the Viatra constraints (cf. discussion in Section 8). For the Dronology system, where we had in-depth knowledge about the system, its properties, and structure considerably less effort was required, whereas, for ROS, we had to familiarize ourselves with the technology, but still were able to create a monitoring platform in less than 20 h, including the process of selecting relevant properties, creating the model, and the constraints. In the Dronology case, for example, this resulted in 8 lines of code that were added manually to the *SuM*, supported by approximately 200 lines of generated Java probe code, about 700 lines of generated platform code, and an additional 2500 LoC generated by EMF for the runtime model. More importantly, when the *SuM* is changed, only minor implementation effort in the *SuM* is required to invoke *GRuM*-generated functions. When properties are added to the model, the entire platform and probes are regenerated.

Table 1
Model-Set-Latency results of the simulation runs (median values). NPR: number of property values collected at runtime/run, SML: time required from event received until set in the runtime model, median, (1st/3rd quartile).

	MoProperty	NPR [#]	SML [ms]
Dronology	Command	470	5.79 (4.42, 6.41)
	FlightControllerData	2,943	5.91 (4.85, 6.52)
	FlightPlan	66	7.43 (5.33, 8.01)
	GPSHealth	6	4.94 (4.32, 7.19)
	HomeLocation	6	11.77 (7.91, 15.46)
	OperationMode	96	5.24 (3.91, 5.74)
	StartupChecks	6	12.86 (6.82, 18.42)
	DroneState	14,702	6.32 (4.99, 6.96)
TurtleBot	AirQuality	1,406	6.60 (5.50, 7.35)
	BatteryStatus	368	6.97 (6.32, 7.63)
	Diagnostic	965	6.66 (5.88, 7.28)
	JointState	1,828	6.35 (5.46, 6.93)
	LaserScan	1,828	6.39 (5.49, 6.96)
	MagneticField	1,828	6.35 (5.40, 6.89)
	Odometry	1,790	6.44 (5.74, 7.06)
	SensorState	1,813	6.50 (5.76, 7.12)
	Velocity	1,829	6.47 (5.62, 7.12)
	VersionInfo	1,822	6.25 (5.55, 6.84)

6.2. RQ2 – System evolution support

In the second part of our evaluation, we specifically focused on the support provided by *GRuM* with regards to monitoring (co-)evolution, and what actions and efforts were required to add monitorable properties to a system, perform constraint checks using the newly collected data, update an existing element (e.g., change the information and/or type of a property) and remove an existing element from the model. We established a set of tasks that impacted all three primary parts of the reference architecture (Rabiser et al., 2019; Pimentel et al., 2012; Cianciaruso et al., 2014; Morrison et al., 2007), focusing on activities related to *Monitoring Setup* and *Execution*. Based on these tasks (cf. Table 2), for each of our two systems, we recorded the different steps necessary for adding, updating, and deleting monitoring properties in the system, modifying the respective constraints, and regenerating the platform.

We specifically examined three changes: (*i*) a new property needs to be monitored and therefore added (A), (*ii*) an existing property is modified (e.g., the property name, MQTT topic, or attributes change) (U), and finally, (*iii*) a property is removed and should no longer be monitored (D). For each change, we documented the steps necessary, the tasks that needed to be performed, the elements/artifacts (e.g., models) that needed to be touched, the tools used, and the degree of manual work/automation provided (cf. Table 2).

Monitoring Setup: All changes performed related to the monitoring setup only require editing one of the two models—the Domain Model Fragment or WM, and all code is subsequently generated. For activities related to the monitoring setup, all changes are performed at the model level, including updating the Domain Model Fragment and the WM. Using the Eclipse Ecore framework (Eclipse, 2021), this is supported by a graphical editor and can be easily achieved with a UML-like diagram. All

Table 2
Key tasks in a monitoring framework based on the monitoring reference architecture by Rabiser et al. (2019). *Aut.* describes the degree of automation support provided by *GRuM*, and *Steps* refers to the number of artifacts that need to be modified for performing a certain action for Adding (A), Updating (U), and Deleting (D) elements from the monitoring platform and SuM.

	Task	Description	Artifacts & Tools	Action required in *GRuM*
Monitoring setup	Monitoring definition	What should be monitored?	Domain Model Frag., (Eclipse Ecore Model and Generator Model) STEPS: 2	**[A|U|D]** MoProperties and MoAgents can be changed in the Domain Model Fragment using a graphical modeling editor. AUT.: PARTIALLY AUTOMATED When the Domain Model Frag. is updated, Eclipse automatically generates required code – no manual code changes required.
	Monitoring instrumentation	How are actual Probes defined/created?	Weaving Model (DSL) Eclipse XText editor STEPS: 1-2	**[A|D]** MoProperties and MoAgents from the Domain Model Frag. need to be linked in the WM and topic bindings need to be established using the DSL. **[U]** If only attributes of MoProperties are updated, no change in the WM. AUT.: PARTIALLY AUTOMATED When WM is updated, the respective code generator can be selected in Eclipse, and the platform and Probes are generated automatically.
Monitoring execution	Monitoring information collection	What is required to collect data from the SuM?	Generated API library (Java: JAR file, Python: module for ROS) STEPS: 1	**[A|U|D]** No extra steps necessary for framework. For instrumented system (Java, Python), the generated library needs to be added and the respective API method calls need to be added. AUT.: FULLY AUTOMATED (PLATF.)/MANUAL (SUM)
	Monitoring information processing	What is required to manage and distribute data in the mon. framework?	–	**[A|U|D]** No extra steps necessary, code for the monitoring platform is generated automatically and a runtime model is instantiated. AUT.: FULLY AUTOMATED
	Monitoring information checking	What is required to define and perform constraint checks?	Viatra editor, Esper rule file STEPS: 1-2	**[A|U|D]** Constraints need to be added/modified either in the Viatra query engine or as new Esper rules. AUT.: MANUAL WITH TOOL SUPPORT.

model code is subsequently generated automatically. Once these changes are applied, the probe API and the monitoring platform are generated automatically. For creating an executable monitoring platform (and the respective runtime model), no changes are required at the code level for either system.

Monitoring Execution: Once respective probe APIs have been generated, they need to be linked with the *SuM*. If agents and/or properties are added or modified, the new API methods need to be called accordingly. When applying our change scenarios to the Dronology system, this required adding/modifying 1 line of code (1 method call) in the system, calling the respective API method when a value is updated. For the ROS/Python example, the effort is similar. First, the ROS node subscriber needs to be started and the corresponding function call in the probe API executed. Besides this, no additional code is required. All connections, topic publishing, and message forwarding is performed automatically by the generated probe and platform code. In the platform itself, no changes or manual steps are required and the runtime model is executed automatically once the platform has been generated.

Based on analyzing the different change scenarios for two different systems, we can conclude that *GRuM* provides a high degree of automation, reducing the manual effort that is required to adopt changes in the *SuM* in the monitoring platform. More importantly, the majority of steps performed are supported by tools and editors, and only the actual instrumentation, i.e. linking the *SuM* to the respective probes, requires actual changes in the source code of the system. None of the changes performed required a manual change in the code of the monitoring platform, and all necessary code could be generated automatically.

6.3. RQ3 – Performance of the generated monitoring platforms

As explained earlier, one question that often arises with code-generated solutions is whether they perform efficiently. This part of the evaluation uses the two deployed monitoring solutions from RQ1 to assess *GRuM*'s performance for monitoring a system, collecting runtime information, and evaluating constraints. In both cases, we collected the following metrics: The number of messages, i.e., monitorable properties sent to the platform (NPR), the latency of the platform (set-model-latency), i.e., time required from setting a value to the runtime model (excluding network delay from the system to *GRuM*) (SML), the number of constraint checks/violations reported (NCST) and time required for performing a constraint check after an element in the model changed (TCST).

The goal of this experiment is to assess whether the generated monitoring platform can handle realistic data from either a high-fidelity simulator (Dronology case) or actual physical hardware (TurtleBot case). We excluded communication latency from our experiments, as this would equally affect any monitoring solution that requires central data analysis. Instead, we focused on the latency when data is available and propagated in the runtime model, and constraint evaluation times, two critical characteristics of the monitoring platform itself.

Case 1 – Dronology: Using Dronology's Software-in-the-loop (SITL) simulator we created a UAV search mission scenario where multiple UAVs search for a missing person in a predefined area. We deployed Dronology with our generated probes and the simulation environment to six Raspberry Pis (RPi 4 with 4 GB of RAM, running Raspian Buster), each representing an individual UAV and connected via Wifi (similar to companion computers

Table 3
Constraints used in the evaluation. NCST describes the median violations reported in our seeded runs, and TCST shows the median evaluation time from when the value was received until the constraint violation was reported in the platform.

CST	Description	Type	NCST [#]	TCST [ms]
D01	*Altitude Restrictions:* The UAV must not exceed a maximum altitude of 50 m	Q	492	0.03
D02	*Speed Restrictions:* The UAV must not exceed a maximum speed of 15 ms/s	Q	492	0.07
D03	*Minimum Battery Level:* The UAV has to maintain a minimum battery voltage of 10.5 V	Q	492	0.02
D04	*Valid Goto-Commands:* A waypoint sent to the UAV must have valid latitude, longitude and altitude values	Q	286	0.09
D05	*Startup Arming Check:* Before active, the UAV needs to compute its arming checks	Q	6	0.30
D06	*Flight Controller Startup:* Before active, the UAV flight controller needs to properly startup	Q	6	0.17
D07	*GPS:* A minimum number of 10 satellites must be available	Q	1458	0.06
D08	*Flight Completion:* An assigned route for a UAV has to be completed within 5 min	T	42	59.00
D09	*Update Frequency:* State data from the UAV, updating its speed and position must be sent at least every 2 s	T	541	58.70
D10	*Geolocation Accuracy:* When GPS accuracy of the UAV is low for more than 30 s, an active mission cannot be continued and manual control has to be assumed	Q/T	353	71.75
T01	*Movement Speed Limit:* To maintain accuracy during navigation, the bot must not move faster than 2.5 m/s	Q	65	0.09
T02	*Minimum Battery Status:* The bot has to maintain a minimum battery level of 5%	Q	65	0.09
T03	*Minimum Battery Voltage:* The bot has to maintain a voltage between 10.5 and 12.5 V	Q	65	0.06
T04	*Power Supply Health:* The bot's power supply must remain in a healthy condition, e.g., no overheating	Q	65	0.05
T05	*Diagnostics Error:* The operating level of the hardware components (actuator etc.) must not be in an error state	Q	49	0.07
T06	*$C0_2$ Limit:* The $C0_2$ measurements from the air quality sensor must not exceed a threshold of 800 ppm	Q	76	0.08
T07	*Obstacle:* The bot must maintain a minimum distance of 5 cm to an object, as detected by the Lidar unit	Q	65	0.08
T08	*Speed Reduction:* When the bot operates below 25% battery level, the speed must not exceed 2.0 m/s anymore	T	88	52.10
T09	*Diagnostics temporal check:* A stale state of the actuator must change within 10 s to another state	T	206	32.58
T10	*Measurements Accuracy*: To ensure accurate measurements, an alert should be raised when measurements are transmitted while moving	T	191	84.00

on real UAVs). The monitoring infrastructure was set up on a standard Desktop Computer, with 16 GB of RAM, running Ubuntu 20, using Java 11, and Ecore 2.23. We randomly assigned 5 flight routes to each UAV, and then collected runtime data and performed constraint checks. We first performed a standard simulation scenario with no anticipated constraint violations, and then in a second simulation, we seeded errors every 30 s in order to evaluate whether violations were reported correctly and in a timely fashion. Both scenarios (with and without errors), were executed 5 times and median values are reported over all 5 runs. An overview of the number of events captured and constraint checks performed is provided in Tables 1 and 3.

Overall, each simulation run lasted about 41 min representing a typical upper bound of a typical battery-powered UAV flight. During this time, a total of 18,295 events were sent to the monitoring platform. The median time to set a property in the model was between 4.9 and 12.9 ms. We only observed higher latencies (>7.5 ms) for the properties that were collected a single time during UAV startup (12.9 ms) (the startup checks and homelocation) with all other properties well below this limit. The median time to evaluate a constraint for the Viatra query engine was between 0.02 ms and 0.9 ms. As Esper is a stream processing engine that only matches a certain pattern, i.e., a constraint violation, we were only able to count and/or measure constraint violations for the seeded errors and not actual executions. In total 4168 violations, for both Esper and Viatra constraints were reported. As Esper constraints are temporal, the time from the point an event is added to the event stream to the time a violation is detected varies depending on the type of constraint. For CST-D08 (cf. Table 1), this means that a constraint violation is triggered after 5 min (300 s), for CST-D08 after 2 s, when the second `DroneState` property is not received in time, and after 30 s for CST-D10 respectively. In both cases our platform handled diverse constraints, some of which are executed very frequently, some of which are only executed once, providing an evaluation result of fewer than 0.9 ms for the Viatra constraint and 71 ms for a timing constraint evaluated with Esper.

Case 2 – ROS TurtleBot: For the second case, we performed a series of runs with two TurtleBot robots (TurtleBot3 Burger, equipped with a RPi 3, running ROS noetic), tasked with measuring $C0_2$ levels in various offices and hallways to detect and report high concentrations. We used the ROS SLAM (Simultaneous Localization and Mapping) node to create a map of the office space and sent the TurtleBots to different offices multiple times using ROS' 2D Navigation Stack. The monitoring infrastructure was set up on a standard Desktop Computer, with 16 GB of RAM, running Ubuntu 20, using Java 11, and Ecore 2.23. Again, we first executed a scenario without any anticipated constraint violations, followed by a second scenario seeded with random errors. Both scenarios (with and without seeded errors), were executed 3 times and we report the median values over the 3 runs. Each run lasted approx. 16 min with 15,477 events collected per run, which were sent to the monitoring platform, resulting in 980 events per minute. The latency to set values in the model ranged between 6.3 ms and 7 ms (cf. Fig. 7).

For the constraint evaluation runs, we obtained very similar results to the previous experiment. The median time to evaluate a constraint for the Viatra query engine was between 0.05 and 0.09 ms and for Esper constraints between 33 ms and 84 ms, with a median of 935 violations reported per run. All results are listed in Tables 1 and 3.

6.3.1. Scalability

To validate the scalability of the generated platform, we evaluated its capability to deal with a large number of properties and agents. For the Dronology case, we virtualized the setup used on the Raspberry Pis in a Docker container and executed a simulation scenario with 25 UAVs flying and reporting data at the same time. For the second case with the TurtleBots, we increased the frequency with which data was sent to the monitoring platform. Initially, properties were sent every second and every five seconds for the battery measurements which were reduced to 0.25 s resulting in a much higher rate of events and constraint checks. For the approx. 46-min Dronology run with 25 UAVs, we

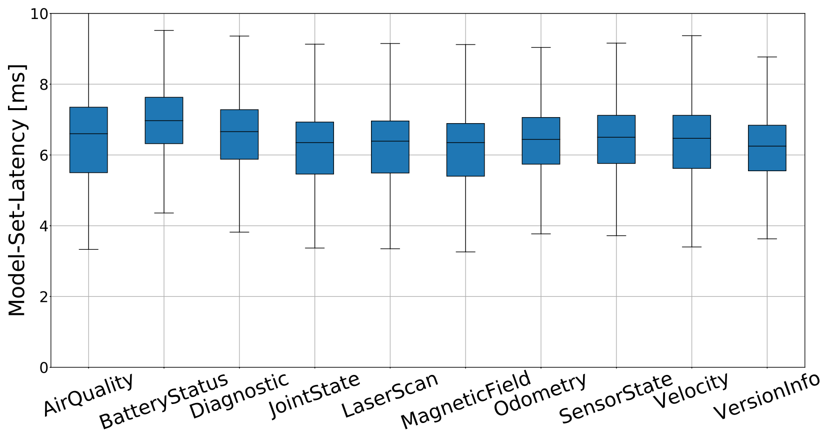


Fig. 7. Model-Set-Latency for Runtime properties for the TurtleBot evaluation runs.

received 71,567 property updates, i.e., more than 1530 property changes per minute, and observed a constant number of messages being received by the platform throughout the run without any backup with comparably low latencies between 4.3 and 6.3 ms. During this run, 20,719 constraint checks were performed with evaluation times between 0.08 and 0.13 ms for the Viatra queries and 46 and 61 ms for the Esper temporal checks.

For the TurtleBot run, we received 64,000 updates during the 16 min run, i.e., approximately 4076 properties per minute. 1173 constraint checks were performed and evaluation times range between 0.07 and 0.1 ms for the Viatra queries and between 63 to 180 ms for the Esper temporal checks.

Analysis of Performance Evaluations: For both cases, we were able to monitor a substantial amount of properties and the platform performed well for updating the runtime model and performing constraint checks. For the Dronology system, we deliberately selected properties with a diverse update frequency, demonstrating that the framework works well with data collected only a few times a minute, or several times a second as for the TurtleBots. Additionally, even when scaling up the number of agents in the Dronology system, and when increasing the frequency at which events are sent to the platform for the TurtleBot robots, we achieved almost constant evaluation and latency times. Furthermore, as all experiments were run on a standard desktop machine, both the latency to update the model and time to perform constraint checks can be further improved when deploying the platform on server hardware. With regards to RQ2, we can conclude that the generated platform is capable of handling a substantial amount of diverse properties from *SuM* using different architecture styles and technologies.

6.4. Comparison to related techniques and trade-offs

In order to contextualize the results of RQ1 and RQ2 and to draw conclusions about the extent to which *GRuM* can reduce required effort, we provide a qualitative comparison against a selection of related approaches. We do not provide quantitative comparisons—as fully recreating each of the other systems and frameworks is out of scope of this paper. For this purpose, we leverage our previous work in the area of analyzing Runtime Monitoring Frameworks (Rabiser et al., 2017) by selecting five (Keller and Ludwig, 2003; Eichelberger and Schmid, 2014b; Kim et al., 2001; Ehlers and Hasselbring, 2011; Inzinger et al., 2013) approaches that (a) cover a broad spectrum of different application domains and application contexts of runtime monitoring; and additionally, (b) provide sufficient information in the paper to allow a comparison with respect to the capabilities provided for Runtime Monitoring and trade-off ease of use and convenience. We structure our discussion along the previously discussed key tasks of “Monitoring Setup”, “Monitoring Execution”, and ‘Monitoring Support” from the monitoring reference architecture (Rabiser et al., 2019). Results from this analysis are summarized in Table 4.

Monitoring Setup: Our *GRuM* framework relies on a strict separation between the description of the *SuM* (and its constituent properties) in the Domain Model Fragment and any information relevant for monitoring in the Weaved Monitoring model. With regards to monitoring definition, similar to many other approaches, we use a textual description based on a dedicated DSL to specify monitoring-related information. Similar to our approach, Java-MaC (Kim et al., 2001) and Inzinger et al. (2013) use their own custom DSLs, and Kieker (Ehlers and Hasselbring, 2011) uses OCL for monitoring definition. In contrast, SPASS-meter and WLSA (Keller and Ludwig, 2003; Eichelberger and Schmid, 2014b) rely on XML files for defining monitoring scopes. Alternatively, SPASS-meter also supports direct annotation of source code, thereby avoiding the need for a separate file for defining monitors. One negative trade-off that we are aware of is that as a precursor to applying *GRuM* the creation of the Domain Model Fragment requires some additional effort. However, as only relevant properties need to be modeled, the perceived effort is relatively low, and the resulting benefits are the tool-assisted creation of the Weaved Monitoring Model (auto-completion, selection of properties, etc.) which greatly eases its creation and maintenance, and the entire framework generated from it. Furthermore, if models are readily available for the system, they can be easily reused and integrated, thus further reducing the effort of this extra step. With regards to the creation of probes, we also follow the paradigm of automatically generating probe code for the target *SuM*. Depending on the approach, either Java bytecode instrumentation (Eichelberger and Schmid, 2014b; Kim et al., 2001) or aspect-oriented programming (Ehlers and Hasselbring, 2011) techniques are employed. With the concept of dedicated *Probe Set Generators*, we introduced one additional layer. This increases flexibility (similar to, for example, Kieker (Ehlers and Hasselbring, 2011) where templates for Python and Java-based systems are provided). PSGs thus facilitate easy reuse, and once a generator for a specific technology/target system has been created it can be easily integrated and readily reused.

Monitoring Execution: Once created, *GRuM* provides a fully operational monitoring framework that can be readily executed. This includes event collection as well as distribution at runtime. While other approaches combine the definition of event monitoring and analysis (e.g., as part of the Event Definition DSL Kim et al., 2001 or SLA metrics Keller and Ludwig, 2003) we again focus on separation of concerns between these two parts. The

Table 4
Comparison of *GRuM* with regards to support and ease of use of key monitoring tasks (Rabiser et al., 2019). ● = fully supported/available; ◑ = partially supported; ○ = not supported/mentioned.

Phase	Framework					
	GRuM	Eichelberger and Schmid (2014b)	Kim et al. (2001)	Ehlers and Hasselbring (2011)	Inzinger et al. (2013)	Keller and Ludwig (2003)
Monitoring setup	●	◑	●	◑	◑	◑
Monitoring execution	●	○	◑	◑	○	○
Monitoring support	◑[a]	◑	○	◑	○	○
Monitoring co-evolution	●	◑	○	○	○	○

[a]Supported via the *GRuM* Monitoring Middleware.

MaC (Kim et al., 2001) architecture, for its Java implementation, for example, relies on a dedicated instrumentor and compiler to manipulate bytecode and a run-time checker. While this allows for some flexibility, and its concepts can also be applied to other languages, it requires a specific implementation of these components. We avoid this by decoupling probes and the actual framework implementation. Kieker (Ehlers and Hasselbring, 2011) on the other hand follows a similar approach as *GRuM*, with a common and extensible monitoring record model for data collection, and a dedicated record consumer model that facilitates runtime checks. As part of the core capabilities in *GRuM*, we leverage the Viatra query engine that again use the generated Ecore code to provide sophisticated tool support for generating model queries/constraints on the monitored properties. More complex constraints can be added via external services connected through the middleware layer.

Monitoring Support: While the main purpose of *GRuM* is the generation of the "base" monitoring framework, the automatically generated Monitoring Middleware enables easy integration of additional services and tools. We have demonstrated this by connecting a CEP engine, which is an approach also used in the framework proposed by Inzinger et al. (2013). The model-to-code transformation, using EMF, *GRuM* generates ready-to-use interfaces and classes and a template Java Maven project, that provides *SuM* specific interfaces and wrapper classes that could significantly ease the task of connecting external applications. While introducing model-based technologies adds to the complexity of *GRuM*, our goal is to "hide" this complexity from the user to the fullest extent possible. Similar to the other monitoring frameworks, we employ a domain-specific language for defining monitoring configurations, ready-to-use code generators, and templates for connecting to external services.

Co-Evolution Support: The aspect of monitoring co-evolution is only lightly addressed by a few other approaches. SPASS-meter briefly addresses the co-evolution aspect, acknowledging that depending on how the scope configurations are defined they may become outdated as the system evolves. When using their external configuration option, specified as XML files, synchronization is required, whereas their inline configurations are directly added to the source code, hence avoiding the synchronization issue. However, the latter requires direct modifications to the source code of the SuM. The results of RQ3, indicate clear benefits when relying on model-based concepts, not only with regards to easy re-generation of the entire framework, but also in terms of the provided support structure. For example, removal or updates of properties from the system/model are immediately reflected in the updated code that is generated (or removed), and error messages and warnings are triggered in the tool-supported editors (e.g., for Viatra model queries that reference a property that no longer exists).

7. Threats to validity

As with any experiment, our evaluation is subject to a number of threats to validity.

Internal validity is related to the rigor of the experimental design. Our evaluation is based on the creation of monitoring models, and the generation of the monitoring platform for two different application scenarios. While these were created by the authors of this paper, the selected properties and resulting constraints were selected from real-world use case scenarios and applications. With regards to the selected systems, for Dronology, we possess in-depth knowledge about its internal design, structure, and functionality which also informed the creation of the Domain Model Fragment and selection of constraints. To avoid bias, we performed the same steps on a second system, the TurtleBot robots where we first had to get familiar with the technology and system characteristics. Additionally, having built a custom, system-specific monitoring solution for one of the systems allowed us to discuss and assess the value of a generic monitoring platform that can easily be applied to different systems and updated when the *SuM* evolves. So far, as part of the evaluation, *GRuM* was used by several authors to create models and generate the Monitoring Platform demonstrating the applicability and evolution efforts. In order to properly assess the usability, and to show broader applicability, we need to involve additional users, e.g., by conducting a dedicated usability study of the framework and provided tools.

External validity refers to the generalizability of results and findings. Our study focused on only two different systems, and therefore, we cannot claim full generalizability of our approach to diverse types of CPS. However, the Dronology system and the TurtleBot systems exhibit significant differences in terms of the technology used, their architectural styles, and properties that are monitored, demonstrating the flexibility of *GRuM* across different types of systems. Further, to minimize the threat of invalid data measurements due to external factors such as OS tasks or interference with other applications, we performed multiple runs of each scenario and calculated the medial values; however, our performance measurements focused on the runtime model and the constraint evaluation and did not include network delays or delays in the underlying *SuM* which likely account for certain latencies such as retrieving the home coordinates of a UAV in Dronology. Most notably, while the model-set-latency and constraint evaluation times are acceptable for the Dronology and TurtleBot systems, we cannot currently guarantee strict real-time deadlines for evaluating constraints. Nevertheless, the modular nature of *GRuM* allows key platform components to be replaced, for example, replacing MQTT with a DDS (DDS Foundation, 2021) middleware implementation providing real-time capabilities.

Construct validity refers to the extent to which a study measures what it claims to be measuring. RQ1 investigated the ability of *GRuM* to describe relevant properties and to generate a customized monitoring platform. We applied *GRuM* to two different systems with real-world applications and physical hardware using properties derived from existing documentation. We were able to represent all identified properties and constraints with our proposed approach and generated a fully functioning, customized monitoring platform, and showed that in both cases, *GRuM*'s generated platform was able to handle realistic amounts of data and different types of properties. While we have been using a simulator for the Dronology use case, Ardupilot is a high-fidelity simulator providing a realistic execution environment, and our previous experience has confirmed that this closely resembles real-world scenarios.

8. Discussion

Results from our evaluation have shown that our model-based *GRuM* framework can be successfully used to (1) describe relevant properties and (2) collect and check runtime data via the automatically generated monitoring platform. The platform was easy to customize through the use of the middleware—which allowed us to seamlessly add the additional temporal constraint engine. Clear separation of concerns makes the platform highly flexible and maintainable.

The distinction between the platform's core capabilities and add-on services allows the core to be regenerated in response to changes in the *SuM*, while the separation between the generation of the platform and the probe set for collecting runtime data enables easy adaptation of *GRuM* for new technologies and types of systems being monitored. In our prototype implementation, we currently use direct instrumentation for the generated Probes (i.e., providing probes that expose an API for submitting new data). This, on the one hand, allows for easy use (similar to a logging API), but on the other hand, requires some manual effort to connect the *SuM* to our framework, and requires source code access. If this is not possible and/or not desired, this can be easily changed by simply replacing the respective Probe Generator with a different one. For example, a new Probe Generator for Java-based systems could use byte-code instrumentation (Havelund and Roşu, 2001), or aspect orientation (Cassar et al., 2017) to collect runtime data. The main advantage however is that the platform can be easily tailored to a new system with no programming effort if a probe Set Generator can be reused from the library or with considerably low effort as shown in RQ1. In comparison, as part of our Dronology system, we have previously implemented a custom monitoring solution specifically tailored to the UAV use case. With a similar technology stack (MQTT, and Drools as a Constraint Rule engine), this required approx. a person-month of effort and roughly 1500 lines of code for a platform that is highly specific and interwoven with the Dronology system. In contrast, *GRuM* required some initial modeling effort, but only minimal implementation effort to call the probe API to connect the system to the platform.

With regards to monitoring co-evolution support, we were able to demonstrate that *GRuM* provides a high degree of automation and reduces the manual effort of modifying code in the monitoring platform, when changes to the *SuM* occur. Certain manual tasks are still required (e.g., adding/modifying properties in the models), but these largely take place at the model level, where tools and support for the users are provided. One aspect we are currently investigating as part of our ongoing work is providing additional support for checking the consistency between the WM and the actual source code of the system, by e.g., annotating Java Classes and fields in the system as monitorable, and subsequently leveraging AST analysis to synchronize, or even automatically generate the WM. Additionally, we are planning on extending our evaluation to include a full user study, providing participants with a system and the requirement to monitor and check certain properties, with the goal of assessing the usability of the platform and modeling approach.

Eclipse and the EMF platform provide a convenient way of defining the Domain Model Fragment and the WM. For the WM, a textual representation via a DSL and the provided EMF editor (including code completion and syntax highlighting) enables easy updates and adaptation of the WM and regeneration of the monitoring platform. The evaluation showed that both Domain Model Fragment and WM can be easily created and updated with minimal effort. However, to further ease the task of preparing the generated platform for a specific *SuM*, we plan to provide additional tool support. While creating Viatra model queries is relatively easy and straightforward, defining temporal constraints as Esper queries is more challenging and time-consuming. We, therefore, plan to further investigate the suitability of other constraint and rule engines and to provide a dedicated DSL with constraint templates that get automatically translated to either Viatra queries or Esper constraints as appropriate.

9. Related work

Related work is primarily in runtime monitoring for CPS, model-based runtime monitoring, and models@runtime.

Runtime Monitoring for CPS: A recent survey by Rabiser et al. (2019) highlights the runtime monitoring area as an active field of research, presenting a reference architecture for monitoring platforms. We based the architecture of *GRuM* on the architecture described in this work. However, the approach of Rabiser et al. is not model-based and does not provide capabilities to generate monitoring support for different types of systems.

Several approaches have been proposed with regards to monitoring CPS and large-scale systems. For example, Doherty et al. (2009) presented a task planning and execution monitoring framework using temporal action logic to specify the behavior of the system. Vierhauser et al. (2016b) presented a case study on monitoring UAVs using their ReMinDs framework. In the domain of self-adaptive systems, Machin et al. (2014) proposed an approach for synthesizing monitored autonomous systems. Kieker (Ehlers and Hasselbring, 2011) provided capabilities for inserting probes for intercepting the system execution and monitoring various runtime aspects of the system. The framework also supports adaptation of monitoring rules for activating and deactivating probes relevant to the current monitoring task. HiFi (Al-Shaer et al., 1999) used programmable agents and filters to manually configure the infrastructure at runtime and adapt different agents. However, while some of these approaches provide support for instrumentation, for example via byte-code instrumentation or predefined monitors (Chen and Rocsu, 2007; Eichelberger and Schmid, 2014a), or customizing the monitoring platform, none of them provide extensive support for automatically generating code or provide an independent-language, applicable across different types of systems and thus lack of a more abstract solution.

Model-based Monitoring & Models@Runtime: In MDE and industrial CPS domain, the term "Digital Twin" has become synonymous with a model of the system instantiated at runtime (Uhlemann et al., 2017; Kirchhof et al., 2020). Specifically, in Models@Runtime research (Blair et al., 2009; Bencomo et al., 2019), the models are instantiated at runtime, and then used to check properties and support self-adaption. However, very few approaches employ model-based concepts to facilitate the generation of a complete monitoring platform. Hili et al. (2020) proposed a model-based architecture for interactive runtime monitoring using model-based techniques. However, while they supported model-to-model transformation as well as automated code generation, their focus was on monitoring real-time and embedded systems. In the domain of robotic applications, MROS by Corbato et al. (2020) supported runtime adaptation of ROS-based systems using a model-based framework. While their approach provides potentially interesting extensions to our ROS PSG, with *GRuM* we aim to provide a more diverse framework that can generate and provide monitoring support for different types of systems. Brand and Giese (2019, 2018) have extensively worked in the area of model-based adaptive monitoring. They proposed a generic adaptive monitoring approach, based on analyzing queries on a runtime model and adjusting periodic or event-driven monitoring tasks. While their work also revolves around runtime models and MDE, their focus is on adaptive monitoring and query support on runtime models, without support for automatically generating the monitoring platform and probes to perform these queries.

Sakizloglou et al. (2020) used runtime models supported by Viatra to check constraints as part of adaptation rules. Búr et al. (2018) provide support for querying CPS runtime properties using distributed graph queries. While we plan to explore distributed monitoring in future work, *GRuM* focuses on the specification, creation, and maintenance of a flexible, and extensible monitoring platform.

Formal approaches for Monitoring & Verification: An entire research area is dedicated to formal approaches and runtime verification with several frameworks that specify monitors using formal models, such as temporal logic or Event Calculus. For example, MOP (Chen and Rocsu, 2007) uses a specification formalism either part of a Java class file or specified independently. It consists of a header containing meta-information, a body specifying the desired property to be checked, and a handler performing certain actions in case the property is violated. Chan et al. (2005) developed a framework for runtime verification of timed and probabilistic non-functional properties in component-based systems. At runtime, the specified assertions can be verified and the framework provides additional capabilities for linking domain-specific models with the system interception code. Drusinsky (2003) have presented Temporal Rover supporting Linear Time Temporal Logic (LTL) including real-time, as well as time series constraints. This facilitates monitoring properties such as stability and calculating sum, average, and temporal minima and maxima. While these approaches provide an important foundation for runtime verification and formal constraint checking, with *GRuM* we further focus on providing support for the entire lifecycle, reducing the burden of creating monitors, and allowing for easy adaptation of constraints, monitors, and instrumentation when the underlying system evolves.

10. Conclusion

In this paper, we present *GRuM*, a novel framework for automatically generating runtime monitors. *GRuM* leverages model-driven technologies for defining a Domain Model Fragment and WM, describing relevant properties and data. Depending on the *SuM*'s technology and architectural style, different types of probes collecting runtime data can be generated. Additionally, *GRuM* automatically generates a customized monitoring platform consisting of a runtime model, a model query engine for defining constraint checks, and a dedicated middleware for connecting external services and applications. We evaluated our approach on two different systems, a UAV management and control system, and a set of TurtleBot 3 robots. The results of our experiments have shown that our implementation of a model-driven framework for automatically generating a runtime monitoring platform is well suited for describing the monitoring needs of non-real-time systems for collecting and checking large amounts of runtime data.

As part of our ongoing future work, we plan to explore ways to support dynamic reconfiguration of the monitoring platform and to provide additional evolution support by synchronizing the models with the *SuM*. Furthermore, we are exploring adding self-adaptation capabilities (Kephart and Chess, 2003) to our *GRuM* framework. Probes then, instead of just sending data to the runtime model, provide an additional back channel to receive and execute adaptation instructions, for example, triggered by the constraint engine when a specific constraint check fails.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

The work described in this paper was partially funded by the Linz Institute of Technology (LIT) LIT–2019–7–INC–316, and by the US National Science Foundation (NSF) Grant CNS:1931962.

References

Aagela, H., Al-Nesf, M., Holmes, V., 2017. An asus_xtion_probased indoor MAPPING using a raspberry pi with turtlebot robot turtlebot robot. In: Proc. of the 23rd Int'L Conf. on Automation and Computing. IEEE, pp. 1–5.
Aizawa, K., Tei, K., Honiden, S., 2018. Identifying safety properties guaranteed in changed environment at runtime. In: Proc. of the 2018 IEEE Int'L Conf. on Agents. IEEE, pp. 75–80.
Al-Shaer, E., Abdel-Wahab, H., Maly, K., 1999. Hifi: A new monitoring architecture for distributed systems management. In: Proc. of the 19th IEEE Int'L Conf. on Distributed Computing Systems. IEEE, pp. 171–178.
Amsters, R., Slaets, P., 2019. Turtlebot 3 as a robotics education platform. In: Proc. of the Int'L Conf. on Robotics in Education. Springer, pp. 170–181.
Asici, T.Z., Karaduman, B., Eslampanah, R., Challenger, M., Denil, J., Vangheluwe, H., 2019. Applying model driven engineering techniques to the development of contiki-based IoT systems. In: Proc. of the 1st Int'L Workshop on Software Engineering Research Practices for the Internet of Things. pp. 25–32.
Baresi, L., Guinea, S., 2005. Towards dynamic monitoring of WS-BPEL processes. In: Proc. of the Int'L Conf. on Service-Oriented Computing. Springer, pp. 269–282.
Baresi, L., Pasquale, L., Spoletini, P., 2010. Fuzzy goals for requirements-driven adaptation. In: Proc. of the Int'L Requirements Engineering Conf.. IEEE, pp. 125–134.
Bencomo, N., Götz, S., Song, H., 2019. Models@run.time: a guided tour of the state of the art and research challenges. Softw. Syst. Model. 18 (5), 3049–3082.
Bergmann, G., Dávid, I., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z., Varró, D., 2015. Viatra 3: A reactive model transformation platform. In: Proc. of the Int'L Conf. on Theory and Practice of Model Transformations. Springer, pp. 101–110.
Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A., 2010. Incremental evaluation of model queries over EMF models. In: Proc. of the Int'L Conf. on Model Driven Engineering Languages and Systems. Springer, pp. 76–90.
Blair, G., Bencomo, N., France, R.B., 2009. Models@ run.time. Computer 42 (10), 22–27.
Brand, T., Giese, H., 2018. Towards software architecture runtime models for continuous adaptive monitoring. In: Proc. of the 13th Int'L Workshop on Models@Run.Time. pp. 72–77.
Brand, T., Giese, H., 2019. Generic adaptive monitoring based on executed architecture runtime model queries and events. In: Proc. of the 13th Int'L Conf. on Self-Adaptive and Self-Organizing Systems. IEEE, pp. 17–22.
Bucchiarone, A., Cabot, J., Paige, R.F., Pierantonio, A., 2020. Grand challenges in model-driven engineering: an analysis of the state of the research. SoSyM 19 (1), 5–13.
Búr, M., Szilágyi, G., Vörös, A., Varró, D., 2018. Distributed graph queries for runtime monitoring of Cyber-Physical Systems. In: Proc. of the Int'L Conf. on Fundamental Approaches To Software Engineering. Springer, Cham, pp. 111–128.
Caldiera, V.R.B.G., Rombach, H.D., 1994. The goal question metric approach. Encycl. Softw. Eng. 528–532.
Cassar, I., Francalanza, A., Aceto, L., Ingólfsdóttir, A., et al., 2017. A survey of runtime monitoring instrumentation techniques. In: Proc. of the 2nd Int'L Workshop on Pre-and Post-Deployment Verification Techniques, PrePost@IFM 2017. pp. 15–28.
Chan, K., Poernomo, I., Schmidt, H., Jayaputera, J., 2005. A model-oriented framework for runtime monitoring of nonfunctional properties. In: Quality of Software Architectures and Software Quality. Springer, pp. 38–52.

Chen, F., Rocsu, G., 2007. Mop: An efficient and generic runtime verification framework. In: Proc. of the 22nd Annual ACM SIGPLAN Conf. on Object-Oriented Programming Systems and Applications. ACM, pp. 569–588.
Cianciaruso, L., Di Forenza, F., Di Nitto, E., Miglierina, M., Ferry, N., Solberg, A., 2014. Using models at runtime to support adaptable monitoring of multi-clouds applications. In: Proc. of the 16th Int'L Symp. on Symbolic and Numeric Algorithms for Scientific Computing. IEEE, pp. 401–408.
Cleland-Huang, J., Vierhauser, M., Bayley, S., 2018. Dronology: an incubator for Cyber-Physical Systems research. In: Proc. of the 40th Int'L Conf. on Software Engineering: New Ideas and Emerging Results. pp. 109–112.
Corbato, C.H., Bozhinoski, D., Oviedo, M.G., van der Hoorn, G., Garcia, N.H., Deshpande, H., Tjerngren, J., Wasowski, A., 2020. MROS: Runtime adaptation for robot control architectures. arXiv preprint arXiv:2010.09145.
DDS Foundation, 2021. Data distribution service middleware. https://www.dds-foundation.org, [Last accessed 01-06-2021].
Del Fabro, M.D., Bézivin, J., Jouault, F., Breton, E., Gueltas, G., 2005a. AMW: a generic model weaver. In: 1 Ere Journées sur l'Ingénierie Dirigée par les Modèles (IDM05). pp. 105–114.
Del Fabro, M.D., Bézivin, J., Jouault, F., Valduriez, P., et al., 2005b. Applying generic model management to data mapping. In: BDA.
Del Fabro, M.D., Bézivin, J., Valduriez, P., 2006. Weaving models with the eclipse AMW plugin. In: Eclipse Modeling Symposium, Eclipse Summit Europe, Vol. 2006. Citeseer, pp. 37–44.
Doherty, P., Kvarnström, J., Heintz, F., 2009. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. Auton. Agents Multi-Agent Syst. 19 (3), 332–377.
Drusinsky, D., 2003. Monitoring temporal rules combined with time series. In: Proc. of the Int'L Conf. on Computer Aided Verification. Springer, pp. 114–117.
Eclipse, 2021. Ecore tools - graphical modeling for ecore. https://www.eclipse.org/ecoretools, [Last accessed 01-06-2022].
Eclipse Foundation, 2021. Xtext language engineering framework. https://www.eclipse.org/Xtext, [Last accessed 01-06-2021].
Ehlers, J., Hasselbring, W., 2011. A self-adaptive monitoring framework for component-based software systems. In: Proc. of the 5th Europ. Conf. on Software Architecture. Springer, pp. 278–286.
Eichelberger, H., Schmid, K., 2014a. Flexible resource monitoring of java programs. J. Syst. Softw. 93, 163–186.
Eichelberger, H., Schmid, K., 2014b. Flexible resource monitoring of java programs. J. Syst. Softw. 163–186.
Eichleay, M., Evens, E., Stankevitz, K., Parker, C., 2019. Using the unmanned aerial vehicle delivery decision tool to consider transporting medical supplies via drone. Glob. Health: Sci. Pract. 7 (4), 500–506.
Elgendi, I., Hossain, M.F., Jamalipour, A., Munasinghe, K.S., 2019. Protecting cyber physical systems using a learned MAPE-K model. IEEE Access 7, 90954–90963.
Espertech, 2021. Esper complex event processing. https://www.espertech.com/esper, [Last accessed 01-06-2021].
Foundaition, E., 2021. Eclipse modeling framework (EMF). https://www.eclipse.org/modeling/emf, [Last accessed 01-06-2021].
France, R., Rumpe, B., 2007. Model-driven development of complex software: A research roadmap. In: Future of Software Engineering (FOSE'07). IEEE, pp. 37–54.
Galster, M., Bucherer, E., 2008. A taxonomy for identifying and specifying non-functional requirements in service-oriented development. In: Proc. of the 2008 IEEE Congress on Services-Part I. IEEE, pp. 345–352.
Goldberg, A., Haveland, K., 2003. Instrumentation of java bytecode for runtime analysis. In: Formal Techniques for Java-Like Programs, no. NAS2-00065.
Hamida, A.B., Bertolino, A., Calabrò, A., De Angelis, G., Lago, N., Lesbegueries, J., 2012. Monitoring service choreographies from multiple sources. In: Proc. of the 4th Int'L Workshop on Software Engineering for Resilient Systems. Springer, pp. 134–149.
Havelund, K., Roşu, G., 2001. Monitoring java programs with java pathexplorer. Electron. Notes Theor. Comput. Sci. 55 (2), 200–217.
Hawkins, R., Habli, I., Kolovos, D., Paige, R., Kelly, T., 2015. Weaving an assurance case from design: a model-based approach. In: Proc. of the 16th Int'L Symp. on High Assurance Systems Engineering. IEEE, pp. 110–117.
Hili, N., Bagherzadeh, M., Jahed, K., Dingel, J., 2020. A model-based architecture for interactive run-time monitoring. Softw. Syst. Mod. 1–23.
Inzinger, C., Satzger, B., Hummer, W., Dustdar, S., 2013. Specification and deployment of distributed monitoring and adaptation infrastructures. In: Proc. of the Service-Oriented Computing-ICSOC 2012 Workshops: ICSOC 2012, International Workshops ASC, DISA, PAASC, SCEB, SeMaPS, WESOA, and Satellite Events, Shanghai, China, November 12-15, 2012, Revised Selected Papers 10. Springer, pp. 167–178.
JBoss, 2021. Roaster java source parser library. https://github.com/forge/roaster, [Last accessed 01-06-2021].
Jouault, F., Vanhooff, B., Bruneliere, H., Doux, G., Berbers, Y., Bézivin, J., 2010. Inter-DSL coordination support by combining megamodeling and model weaving. In: Proc. of the 2010 ACM Symp. on Applied Computing. pp. 2011–2018.
Keller, A., Ludwig, H., 2003. The WSLA framework: Specifying and monitoring service level agreements for web services. J. Netw. Syst. Manage. 57–81.
Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Computer 36 (1), 41–50.
Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan, M., 2001. Java-MaC: a run-time assurance tool for Java programs. Electron. Notes Theor. Comput. Sci. 55 (2), 218–235.
Kirchhof, J.C., Michael, J., Rumpe, B., Varga, S., Wortmann, A., 2020. Model-driven digital twin construction: synthesizing the integration of Cyber-Physical Systems with their information systems. In: Proc. of the 23rd ACM/IEEE Int'L Conf. on Model Driven Engineering Languages and Systems. pp. 90–101.
Koubâa, A., Sriti, M.-F., Javed, Y., Alajlan, M., Qureshi, B., Ellouze, F., Mahmoud, A., 2016. Turtlebot at office: A service-oriented software architecture for personal assistant robots using ros. In: Proc. of the 2016 Int'L Conf. on Autonomous Robot Systems and Competitions. IEEE, pp. 270–276.
Kritzinger, W., Karner, M., Traar, G., Henjes, J., Sihn, W., 2018. Digital twin in manufacturing: A categorical literature review and classification. IFAC-PapersOnLine 51 (11), 1016–1022.
Li, B., Ji, S., Liao, L., Qiu, D., Sun, M., 2013. Monitoring web services for conformance. In: Proc. of the 7th Int'L Symp. on Service Oriented System Engineering. IEEE, pp. 92–102.
Machin, M., Dufossé, F., Blanquart, J.-P., Guiochet, J., Powell, D., Waeselynck, H., 2014. Specifying safety monitors for autonomous systems using model-checking. In: Proc. of the 33rd Int'L Conf. on Computer Safety. Springer, pp. 262–277.
Maiden, N., 2013. Monitoring our requirements. IEEE Softw. 30 (1), 16–17.
Mansouri-Samani, M., Sloman, M., 1993. Monitoring distributed systems. IEEE Netw. 7 (6), 20–30.
Martínez-Fernández, S., Vollmer, A.M., Jedlitschka, A., Franch, X., López, L., Ram, P., Rodríguez, P., Aaramaa, S., Bagnato, A., Choraś, M., et al., 2019. Continuously assessing and improving software quality with software analytics tools: a case study. IEEE Access 7, 68219–68239.
Mayr-Dorn, C., Winterer, M., Salomon, C., Hohensinger, D., Ramler, R., 2021. Considerations for using block-based languages for industrial robot programming-a case study. In: Proc. of the 2021 IEEE/ACM 3rd Int'L WS on Robotics Software Engineering. IEEE, pp. 5–12.
Morrison, R., Balasubramaniam, D., Oquendo, F., Warboys, B., Greenwood, R.M., 2007. An active architecture approach to dynamic systems co-evolution. In: European Conference on Software Architecture. Springer, pp. 2–10.
MQTT, 2023. MQTT message broker. https://mqtt.org, [Last accessed 01-06-2021].
Nikolakis, N., Maratos, V., Makris, S., 2019. A cyber physical system (CPS) approach for safe human-robot collaboration in a shared workplace. Robot. Comput.-Integr. Manuf. 56, 233–243.
Open Robotics, 2023. Gazebo. https://gazebosim.org/home, [Last accessed 01-01-2023].
Pimentel, J., Castro, J., Santos, E., Finkelstein, A., 2012. Towards requirements and architecture co-evolution. In: Proc. of the Int'L Conf. on Advanced Information Systems Engineering. Springer, pp. 159–170.
Popescu, R., Staikopoulos, A., Clarke, S., 2010. An extensible monitoring and adaptation framework. In: Proc. of the 2009 ICSOC/ServiceWave Workshops. Springer, pp. 314–324.
PX4, 2021. Open source flight controller. https://px4.io, [Last accessed 01-01-2023].
Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y., et al., 2009. ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software, Vol. 3. Kobe, Japan, p. 5.
Quiñones, E., Royuela, S., Scordino, C., Gai, P., Pinho, L.M., Nogueira, L., Rollo, J., Cucinotta, T., Biondi, A., Hamann, A., Ziegenbein, D., Saoud, H., Soulat, R., Forsberg, B., Benini, L., Mando, G., Rucher, L., 2020. The AMPERE project: A model-driven development framework for highly parallel and EneRgy-efficient computation supporting multi-criteria optimization. In: Proc. of the 23rd Int'L Symp. on Real-Time Dist. Comp.. pp. 201–206. http://dx.doi.org/10.1109/ISORC49007.2020.00042.
Rabiser, R., Guinea, S., Vierhauser, M., Baresi, L., Grünbacher, P., 2017. A comparison framework for runtime monitoring approaches. J. Syst. Softw. 125, 309–321.
Rabiser, R., Schmid, K., Eichelberger, H., Vierhauser, M., Guinea, S., Grünbacher, P., 2019. A domain analysis of resource and requirements monitoring: Towards a comprehensive model of the software monitoring domain. Inf. Softw. Technol. 111, 86–109.
Reynolds, O., García-Domínguez, A., Bencomo, N., 2020. Towards automated provenance collection for runtime models to record system history. In: Proc. of the 12th System Analysis and Modelling Conf.. pp. 12–21.
Robotis, 2021. ROBOTIS e-Manual for TurtleBot3. [Last accessed 01-01-2023], URL https://emanual.robotis.com/docs/en/platform/turtlebot3/overview.
Ruz, C., Baude, F., Sauvan, B., 2010. Enabling SLA monitoring for component-based SOA applications–a component-based approach. In: Proc. of the Work in Progress Session SEAA. pp. 41–42.

Sakizloglou, L., Ghahremani, S., Brand, T., Barkowsky, M., Giese, H., 2020. Towards highly scalable runtime models with history. In: Proc. of the IEEE/ACM 15th Int'L Symp. on Software Engineering for Adaptive and Self-Managing Systems. ACM, pp. 188–194.
Schörner, M., Wanninger, C., Hoffmann, A., Kosak, O., Reif, W., 2021. Architecture for Emergency Control of Autonomous UAV Ensembles. In: Proc. of the 2021 IEEE/ACM 3rd Int'L Workshop on Robotics Software Engineering. pp. 41–46.
Shakhatreh, H., Sawalmeh, A.H., Al-Fuqaha, A., Dou, Z., Almaita, E., Khalil, I., Othman, N.S., Khreishah, A., Guizani, M., 2019. Unmanned aerial vehicles (UAVs): A survey on civil applications and key research challenges. IEEE Access 7, 48572–48634.
2021. Software engineering datasets for small unmanned aerial systems. https://github.com/SAREC-Lab/sUAS-UseCases, [Last accessed 01-01-2023].
Stadler, M., Vierhauser, M., Garmendia, A., Wimmer, M., Cleland-Huang, J., 2022. Flexible model-driven runtime monitoring support for cyber-physical systems. In: Proc. of the 2022 IEEE/ACM 44th Int'L Conf. on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 350–351.
Sykes, D., Keighren, G., 2018. Industrial-scale environments with bounded uncertainty: a productivity maximisation challenge. In: Proc. of the 2018 IEEE/ACM 1st Int'L WS on Robotics Software Engineering. IEEE, pp. 29–32.
Uhlemann, T.H.-J., Lehmann, C., Steinhilper, R., 2017. The Digital Twin: Realizing the Cyber-Physical Production System for Industry 4.0. Procedia Cirp. 61, 335–340.
Vierhauser, M., Marah, H., Garmendia, A., Cleland-Huang, J., Wimmer, M., 2021a. Towards a model-integrated runtime monitoring infrastructure for Cyber-Physical Systems. In: Proc. of the 2021 IEEE/ACM 43rd Int'L Conf. on Software Engineering: New Ideas and Emerging Results. IEEE, pp. 96–100.
Vierhauser, M., Md Nafee, A.I., Agrawal, A., Cleland-Huang, J., Mason, J., 2021b. Hazard analysis for human-on-the-loop interactions in sUAS systems. In: Proc. of the 29th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. pp. 8–19.
Vierhauser, M., Rabiser, R., Grünbacher, P., 2016a. Requirements monitoring frameworks: A systematic review. Inf. Softw. Technol. 80, 89–109.
Vierhauser, M., Rabiser, R., Grünbacher, P., Seyerlehner, K., Wallner, S., Zeisel, H., 2016b. ReMinds: A flexible runtime monitoring framework for systems of systems. J. Syst. Softw. 112, 123–136.
Wang, Y., McIlraith, S.A., Yu, Y., Mylopoulos, J., 2009. Monitoring and diagnosing software requirements. Autom. Softw. Eng. 3–35.
Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M., 2013. On patterns for decentralized control in self-adaptive systems. In: Software Engineering for Self-Adaptive Systems II. Springer, pp. 76–107.
Zavala, E., Franch, X., Marco, J., 2019. Adaptive monitoring: A systematic mapping. Inf. Softw. Technol. 105, 161–189.

Michael Vierhauser is a senior researcher at the LIT Secure and Correct Systems Lab at the Johannes Kepler University Linz, Austria. He holds a Master's degree in Software Engineering and Ph.D. in Computer Science from the Johannes Kepler University Linz. His current research interests include Cyber-Physical Systems, Safety Assurance, and Runtime Monitoring.

Antonio Garmendia is an assistant professor at the Universidad Autónoma de Madrid (UAM) and is a member of the "Modelling and Software Engineering" research group (http://www.miso.es). He received his Ph.D. in Computer and Telecommunication Engineering from the UAM. As a Ph.D. student, he made a research visit to the PhilippsUniversity Marburg (Germany). He was a postdoctoral researcher at the WIN-SE department JKU Linz, from January 2020 to August 2022. His research interests are in scalability in Model-Driven Engineering (MDE) and the construction of graphical modeling environments. He has participated in the MONDO EU project on scalability in MDE.

Marco Stadler is studying Business Informatics in the master's program at Johannes Kepler University Linz, with a focus on Networks & Security and Software Engineering. Prior to his time at JKU, he attended the bachelor's program in Business Informatics at the University of Regensburg, Germany. Besides his studies, Marco gained experience as a software developer in the IT finance industry. His research interests focus on runtime monitoring and Cyber-Physical Systems.

Manuel Wimmer is a Full Professor and Head of the Department of Business Informatics – Software Engineering at Johannes Kepler University Linz. His research interests include Software Engineering, ModelDriven Engineering, and Cyber-Physical Systems. Find out more at https://se.jku.at/manuel-wimmer.

Jane Cleland-Huang is a Professor and Chair in the Department of Computer Science and Engineering at the University of Notre Dame. Her research interests focus upon Software and Systems Traceability for Safety-Critical Systems with a particular emphasis on the application of machine learning techniques to solve large-scale software and requirements engineering problems. She is the lead PI of the DroneResponse project, Chair of the IFIP 2.9 Working Group on Requirements Engineering, and Associate Editor of the Communications of the ACM. Recent organizational roles have included Program Chair of the 2020 2 International Conference on Software Engineering and General Chair of the 2021 International Requirements Engineering Conference.