Towards a Model-Integrated Runtime Monitoring Infrastructure for Cyber-Physical Systems

Michael Vierhauser, Hussein Marah, Antonio Garmendia
Johannes Kepler University Linz
michael.vierhauser@jku.at

Jane Cleland-Huang
University of Notre Dame
South Bend, IN, USA
janeclelandhuang@nd.edu

Manuel Wimmer
CDL-MINT, Johannes Kepler University Linz
Linz, Austria
manuel.wimmer@jku.at

***Abstract*—Runtime monitoring is essential for ensuring the safe operation and enabling self-adaptive behavior of Cyber-Physical Systems (CPS). It requires the creation of system monitors, instrumentation for data collection, and the definition of constraints. All of these aspects need to evolve to accommodate changes in the system. However, most existing approaches lack support for the automated generation and setup of monitors and constraints for diverse technologies and do not provide adequate support for evolving the monitoring infrastructure. Without this support, constraints and monitors can become stale and become less effective in long-running, rapidly changing CPS. In this "new and emerging results" paper we propose a novel framework for model-integrated runtime monitoring. We combine model-driven techniques and runtime monitoring to automatically generate large parts of the monitoring framework and to reduce the maintenance effort necessary when parts of the monitored system change. We build a prototype and evaluate our approach against a system for controlling the flights of unmanned aerial vehicles.**

***Index Terms*—Runtime Monitoring, Cyber-Physical Systems, Model-Driven Engineering, Evolution**

I. Introduction

Software-intensive systems, including Cyber-Physical Systems (CPS), such as autonomous vehicles [1], self-adaptive unmanned aerial vehicles (UAVs) [2]–[4], and factory floor robots [5], are becoming increasingly ubiquitous in society. They introduce safety concerns, which are typically addressed through regulatory and safety requirements. For example, UAVs must avoid no-fly zones, comply with speed limits, and maintain sufficient battery power. The verification and validation of a CPS thus heavily relies on support for runtime monitoring of its behavior and properties [6].

However, establishing support for runtime monitoring is a non-trivial task. Runtime information must be collected via instrumentation, or by specifying respective data-buses, and then, constraints need to be defined in order to detect deviations at runtime. These deviations must be brought to the attention of the user through alerts and/or visualizations that depict the system status. This often requires customized solutions, especially when off-the-shelf approaches only cover a subset of properties that need to be monitored and checked.

The situation is exacerbated when the system evolves and the instrumentations, monitors, and constraints become outdated. Thus, a monitoring infrastructure needs to co-evolve with the system being monitored to ensure the validity and correctness of the constraint checking results. Keeping the monitoring environment synchronized with the evolving system can be a tedious and time-consuming endeavor; however, neglecting to do so can lead to decreasing effectiveness as constraints start producing false-positive errors and other important constraints are missing.

The Model-driven Engineering (MDE) paradigm [7], provides methods for addressing evolution and maintenance issues related to software systems. In such a scenario, changes are performed solely to the model and system code is generated automatically, without the need to manually adapt and modify different parts of the system. In this sense, it seems promising to employ the same technique to model a monitoring framework, define events, data that needs to be collected, and link constraints to specific elements in the model [8], [9].

Therefore, the goal of this work is to apply MDE techniques to automatically generate a monitoring environment *without the need for a complete model of the monitored system*. In this new and emerging results paper, we make the following contributions: (1) we propose a conceptual framework for event-based, model-integrated monitoring. *ModIRMo* provides support for modeling relevant parts of the system, automatically generating a Monitoring API to collect events and data from the system, instantiating the model at runtime to create a "minimal viable" Digital Twin, and defining and checking constraints on the runtime model; (2) we provide a prototype implementation of *ModIRMo* and perform an initial evaluation with a CPS in the UAV domain; (3) we identify research areas that will drive the development of our ongoing work.

II. Motivating Example & Related Work

Runtime monitoring plays a crucial role in ensuring that a system operates safely and adheres to its specified performance, safety, and other types of constraints. For example, in 2017 a UAV was observed by a helicopter pilot flying far above the UAV's legal limits. The UAV's failsafe mechanism had activated the automated RTL (return to launch) [10], causing it to ascend to over 700 feet above ground level. While the UAV pilot had maintained legal flying altitudes during manual flight, the failsafe RTL altitude had been set above the legal limits for UAVs in the US. With proper runtime checks, this incident may have been avoided either by issuing warnings when the UAV approached the maximum legal altitude or during pre-launch checks of the failsafe RTL altitude property.

Runtime monitoring represents an active area of research as summarized in a recent survey by Rabiser *et al.* [11]. In work on Models@Runtime [12], [13], models are instantiated at runtime, used to check properties of the system, and support self-adaption of the system. However, very few approaches rely on model-based concepts. One major challenge for the successful application of runtime monitoring stems from the *significant upfront investment in implementing, setting up, and maintaining the monitoring infrastructure, and creating respective system instrumentation to collect runtime data*. While some approaches provide partial support, for example via byte-code instrumentation or predefined monitors [14], [15], there is a lack of a more abstract and language-independent solution.

In the MDE and industrial CPS domain, the term "Digital Twin" has become synonymous with a model of the system instantiated at runtime [16], [17]. However, approaches still lack support for system instrumentation or efficient application of model-driven techniques, such as model transformation, or automated code generation. Another aspect that is often neglected is system evolution. Systems in general and CPS in particular are subject to constant change at both the software and hardware levels. Such changes often affect the monitoring infrastructure, as existing instrumentation is rendered incompatible or constraints defined on events and data become stale. Few approaches provide support for automatically updating or modifying monitors or constraints [18]. However, in order for runtime monitoring to be used efficiently throughout the lifetime of a system, *changes in the system need to be reflected in the runtime monitoring infrastructure, and the monitoring infrastructure needs to co-evolve with the system*.

Model-driven Engineering can play a pivotal role in supporting and improving monitoring solutions as it provides concepts that address the needs for monitoring CPS, deals with diverse types of artifacts and constraints, and evolves monitors as the CPS evolves. In the following section, we present *ModIRMo*, our model-integrated framework for supporting runtime monitoring of CPS.

III. Approach

ModIRMo has four major components that facilitate the automated generation of monitors, APIs, and the runtime model. These components provide coverage of the main elements of our previously derived reference architecture for monitoring systems [11]. The four parts cover (1) the Monitoring Meta-Model and the Domain Model; (2) collecting events and data from the monitored system; (3) data aggregation and distribution; and finally (4) the runtime monitoring infrastructure with support for constraint evaluation.

Monitoring Meta-Model & Domain Model: *ModIRMo* relies on MDE concepts to generate monitoring components but does not require a complete model of the monitored system. The *Monitoring Meta-Model*, depicted in Fig. 2, provides the foundation for our model-integrated runtime monitoring approach. The four main elements are: the `MoSystem` describing the system that is monitored and which in turn contains a number of `MoAgents`. MoAgents can be dynamically instantiated at runtime. They can represent system components, machinery, or individual robots in a CPS. MoAgents in turn exhibit certain monitorable properties (`MoProperties`) describing complex elements such as the current state of a UAV, the operational status of a machine (consisting of several values), or simple values `MoValue` that can be monitored at runtime.

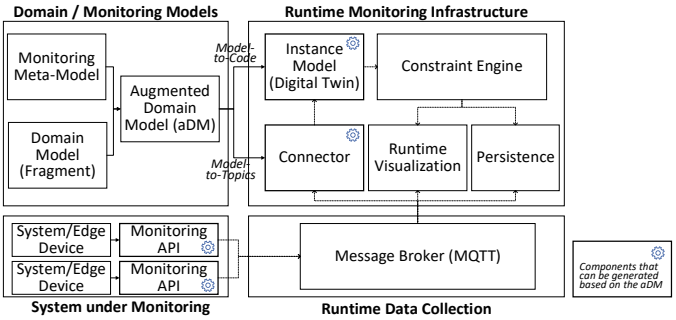

Fig. 1: Overview of *ModIRMo*'s main components. The highlighted components are generated based on the aDM.

A new monitoring system is created by instantiating this Monitoring Meta-Model and creating a Domain Model Fragment. This model only needs to describe parts of the system that will be monitored (i.e., agents and their properties), and not the entire system. UML stereotypes are then used to augment the Domain Model with monitoring information (i.e., describing which parts represent `MoAgents` and `MoProperties`) in order to link the Monitoring Meta-Model to the Domain Model Fragment. The Augmented Domain Model (aDM) provides all the information needed to instrument the system and to synthesize a monitoring environment using Model to Code Transformation. Fig. 3 shows a partial version of the aDM for monitoring a UAV system with further details provided in Section IV.

Runtime Data Collection: Various approaches have been proposed to instrument systems with probes for collecting runtime information [19]. Examples include byte-code instrumentation, aspect-orientation, and information retrieval from existing service buses. *ModIRMo* generates a customized *MonitoringAPI*, similar to a logging component, based on the aDM. In contrast to generic logging APIs, containing methods such as LOGGER.info(), LOGGER.error(), it provides dedicated methods for each MonitorableProperty. The ability to directly call the respective MonitoringAPI method, eases the task of retrieving correct information; however, alternate types of MonitoringAPIs and automated instrumentation, will be explored in future work as discussed in our roadmap.

Message Broker: The Transportation Layer [11], which is responsible for sending messages from the instrumented system to the component that analyzes the data, is realized as an MQTT Message Broker. MQTT uses hierarchically structured topics to distinguish between different types of information and to reduce the amount of information sent to the *Runtime Monitoring Component*. Only those properties of a specific MoAgent that have changed are sent to the respective topic (e.g., uav1/dronestate) to update the State of UAV1.

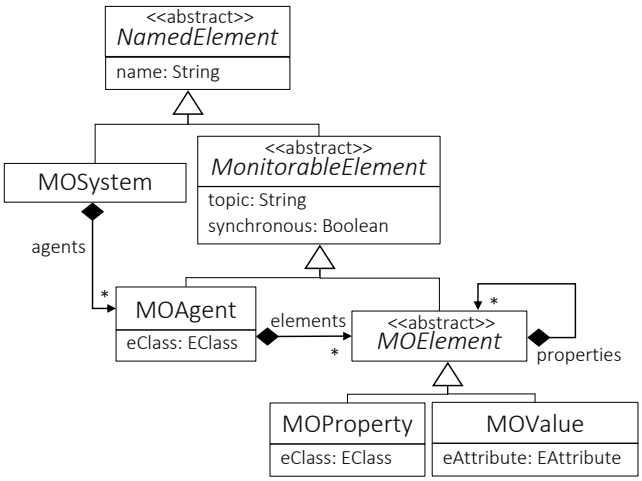

Fig. 2: The Monitoring Meta-Model used in *ModIRMo* for specifying parts of the monitored system.

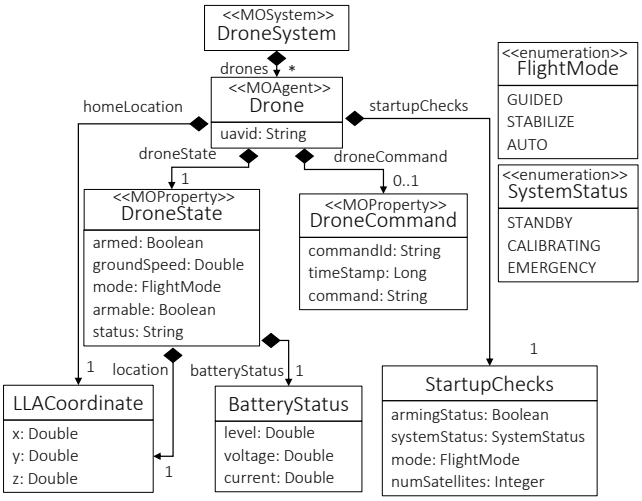

Fig. 3: A partial overview of the Augmented Domain Model (aDM) for the Dronology UAV System.

Leveraging the aDM and the topic information encoded in the model, the entire communication process, including topics, subscriptions, and messages, can be generated automatically.

Runtime Monitoring Infrastructure: At runtime, the aDM can be instantiated to act as a model@runtime or a Digital Twin of the system. Information retrieved from the Message Broker is used to update the runtime instance of the model. The final vital part of a runtime monitoring infrastructure are constraints that need to be evaluated at runtime in order to inspect the system behavior. The MDE community has developed several different constraint languages that can be used at design time or runtime. Constraints can be directly attached to the aDM, at different levels of granularity. System-wide constraints can be applied at the MoSystem level, constraints on individual or multiple agents are applied at the MoAgent level, and constraints regarding specific properties are applied at the MoProperty level. Additionally, components for runtime visualization and for persisting events and runtime data can be connected to the Message Broker by subscribing to the respective topics.

IV. Preliminary Evaluation

To demonstrate feasibility of our monitoring framework and the automated generation of the different components based on the aDM, we developed a prototype and conducted an experimental evaluation. We created a Domain Model for the UAV control system of Dronology [2], defined constraints, and performed simulations with multiple UAVs sending data to the monitoring framework and checking constraints at runtime.

Prototype Implementation: Our *ModIRMo* prototype was based on the Eclipse Modeling Framework (EMF) [20] using Ecore models for the Monitoring Meta-model, the Domain Model Fragment, and the aDM. EMF allowed us to automatically generate Java classes based on the Ecore model, hence the Digital Twin (i.e., the runtime model) was generated automatically. We used Xtend [21], integrated in Eclipse, to support model-to-code transformation to generate Dronology-specific Monitoring API code, the Connector code for retrieving information from the Message Broker and for updating the runtime model, and the glue code, directly from the Ecore model. Finally, we used the VIATRA Validation Framework [22] to define and evaluate constraints. VIATRA provides support to incrementally and reactively perform queries on models, which is beneficial in terms of performance [23]. As a MQTT Broker we used the Mosquitto Message Broker. Runtime visualization support is not currently supported in our prototype but will be added in the future.

Construction of the Domain Model: In the first part of our evaluation we investigated whether the approach would support the modeling and monitoring of critical constraints and the effort required to create the different system specific components (the aDM, Monitoring API, and Connector). For this purpose we used Dronology, a UAV management and control system that supports the flights of multiple physical or simulated drones [2]. Dronology is available as a research environment and we analyzed the source code and published descriptions (e.g., [24]–[26]), to identify components that provided useful runtime monitoring information. The first evaluation focused on modeling the main elements, i.e., UAVs representing MoAgents, modeled as `Drones`, and the `DroneState` and `DroneCommands` sent to the individual UAVs representing MoProperties. The DroneState contains runtime information about the UAV (e.g., its location, speed, battery status) and is constantly updated at runtime. We extended this initial model for the runtime constraint evaluation (cf. below) by adding additional fields representing attributes of the UAV's flight controller for startup checks. The (partial) Dronology aDM is depicted in Fig. 3.

```
pattern startupCheckGPS(d : Drone, sat :java Integer)
Drone.startupchecks.num_satellites(d, sat);
  check(sat < 10);
```

Listing 1: Example of a VIATRA constraint for checking the number of satellites (It is violated if *check* evaluates to true)

While this only represents a fraction of the entire Dronology system, it allowed us to generate relevant components for monitoring UAVs at runtime and check associated constraints.

Based on the aDM we were able to generate the MonitoringAPI code (in Java), the runtime model, as well as code necessary for initiating constraint checks. After updating the initial model, the code was regenerated without the need to manually update the data collection, serialization, or message broker code, and the additional methods in the MonitoringAPI were easily deployed to the Dronology system.

Constraint Definition & Evaluation: In the second part of the evaluation we explored the runtime aspects of our approach by investigating the ability of the prototype to define and check constraints. We performed a series of simulations with Dronology with the previously created aDM. Based on potential drone incidents reported in the literature, we derived nine different constraints (cf. Table I) related to different aspects of operating UAVs. All constraints were written in the VIATRA, supported by an editor that allowed direct access to the elements of the adM. All constraints took 1-2 lines of code (cf. Lst. 1), with the exception of the geofence constraint which required Java code to calculate distances. It took only a few minutes of effort to add each new constraint.

We performed three simulation runs, each approximately 60 minutes, with five UAVs, each assigned five random routes with multiple waypoints. We collected the number of constraint evaluations performed (71,400), the time required from setting a value to the model (excluding network delay from the system to *ModIRMo*) (1.58ms), and the time required for performing constraint checks after an element in the model changed (0.56ms). The reported values show the average of the simulation runs. Based on these initial results we can conclude that *ModIRMo* is capable of dealing with a high number of events and constraint checks. In addition to the simulation runs, to validate the constraints, we simulated a number of misuse cases [27] where we intentionally violated the defined constraints in order to confirm the occurrence of constraint violations. For example, we set the waypoint altitude of a UAV above its maximum value as specified in the constraint, performed longer simulations to trigger battery level violations and sent UAVs to waypoints outside the specified geofence radius. For all nine constraints, we were able to observe violations when the misuse case was executed.

Constraint	Description
Altitude restriction	The UAV must adhere to altitude limits defined for a certain airspace
Safe RTL	The UAV must maintain sufficient battery level to ensure safe return/landing
Flight Mode	The UAV must be in FlightMode “AUTO” mode when flown within the simulation system
Speed Limit	The UAV must not exceed the maximum groundspeed of 10m/s
Geofence Limitation	The UAV must not exceed a maximum distance from its takeoff location
Pre-Checks finished	The UAV system must have finished its pre-checks and calibrations and enter “STANDBY” before take-off
GPS Signal	The UAV must have a fix on a minimum number of 10 GPS satellites
UAV Take-off Mode	The UAV must be in FlightMode “STABILIZE” mode before takeoff while on the ground
Safety Checks	The UAV must have completed all its internal safety checks and have set its state to “armed” before takeoff.

TABLE I: UAV Constraints used with Dronology simulations

V. Research Roadmap & Outlook

The current *ModIRMo* approach requires a human to manually maintain the domain model from which the monitoring environment is generated and regenerated. In future work we plan to explore the use of code annotations that tag classes and constituent properties with *Agent* or *Property* tags to match elements in the Meta-Model. These annotations could be parsed to create the Ecore model.

Constraint Evaluation: The constraints used in the evaluation were related to checking properties of the UAV at runtime but did not include more complex or temporal properties. Numerous constraint languages and evaluation engines have been proposed for performing runtime checks. While VIATRA performed well in our initial evaluation, we will investigate other constraint definition approaches such as Temporal EMF [28], Epsilon, or Complex Event Processing [29] to deal with more diverse constraints. For example, detecting faster than expected battery drain, or fluctuating pitch and roll, could only be achieved with temporal constraints which are not directly supported by VIATRA. Additionally, we will investigate decentralized constraint evaluation [30], [31] where certain constraints can be locally checked on the UAV or an edge device, with the benefits of faster reaction times.

Automated Deployment & Configuration: We will provide better integration with the DevOps process, for example, by supporting automated configuration and deployment of the message broker and runtime configuration of the MonitoringAPI according to constraints required in the current environment. Furthermore, by modeling variabilities of the monitoring environment as a feature model *ModIRMo* could adapt to the different system configurations.

Self-Adaptation Capabilities: Finally, we plan to provide self-adaptation capabilities for CPS. In its current version *ModIRMo* can provide runtime in formation about the system; however, we will leverage this information to allow the system to adapt when certain conditions are met or constraints violated. This form of goal-driven adaptability has been demonstrated by several researchers [32], [33].

In conclusion, the evaluation, supported by our prototype, has shown that *ModIRMo* is capable of handling a large number of events from multiple agents in a CPS. However, additional work is required to refine the Monitoring Meta-Model and to apply *ModIRMo* to a broader range of systems and more diverse programming languages.

Acknowledgments

The work was partially funded by the Linz Institute of Technology, the US National Science Foundation (SHF:1741781 and CPS:1931962), and the Austrian Federal Ministry for Digital and Economic Affairs, and the National Foundation for Research, Technology and Development (CDG).

References

[1] A. Stocco, M. Weiss, M. Calzana, and P. Tonella, "Misbehaviour prediction for autonomous driving systems," in *Proc. of the ACM/IEEE 42nd Int'l Conf. on Software Engineering*, 2020, pp. 359–371.

[2] J. Cleland-Huang, M. Vierhauser, and S. Bayley, "Dronology: an incubator for cyber-physical systems research," in *Proc. of the 40th Int'l Conf. on Software Engineering: New Ideas and Emerging Results*, 2018, pp. 109–112.

[3] V. V. Klemas, "Coastal and environmental remote sensing from unmanned aerial vehicles: An overview," *Journal of Coastal Research*, vol. 31, no. 5, pp. 1260–1267, 2015.

[4] E. Pereira, R. Bencatel, J. Correia, L. Félix, G. Gonçalves, J. Morgado, and J. Sousa, "Unmanned air vehicles for coastal and environmental research," *Journal of Coastal Research*, pp. 1557–1561, 2009.

[5] S. Harapanahalli, N. O. Mahony, G. V. Hernandez, S. Campbell, D. Riordan, and J. Walsh, "Autonomous navigation of mobile robots in factory environment," *Procedia Manufacturing*, vol. 38, pp. 1524–1531, 2019.

[6] H. Shakhatreh, A. H. Sawalmeh, A. Al-Fuqaha, Z. Dou, E. Almaita, I. Khalil, N. S. Othman, A. Khreishah, and M. Guizani, "Unmanned aerial vehicles (UAVs): A survey on civil applications and key research challenges," *IEEE Access*, vol. 7, pp. 48 572–48 634, 2019.

[7] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice, Second Edition*, ser. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017. [Online]. Available: https://doi.org/10.2200/S00751ED2V01Y201701SWE004

[8] A. Tundo, M. Mobilio, M. Orrù, O. Riganelli, M. Guzmàn, and L. Mariani, "Varys: An agnostic model-driven monitoring-as-a-service framework for the cloud," in *Proc. of the 2019 27th ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering*, 2019, pp. 1085–1089.

[9] M. Kintz, M. Kochanowski, and F. Koetter, "Creating user-specific business process monitoring dashboards with a model-driven approach." in *MODELSWARD*, 2017, pp. 353–361.

[10] Staff reporter, "Pilot of drone that nearly hit chp helicopter says it was on autopilot," *CBS SF Bay Area News Outlet*, pp. URL: https://tinyurl.com/UAVIncident–2, (Last accessed 07/20/20), 2015.

[11] R. Rabiser, K. Schmid, H. Eichelberger, M. Vierhauser, S. Guinea, and P. Grünbacher, "A domain analysis of resource and requirements monitoring: Towards a comprehensive model of the software monitoring domain," *Information and Software Technology*, vol. 111, pp. 86–109, 2019.

[12] G. Blair, N. Bencomo, and R. B. France, "Models@run. time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[13] N. Bencomo, S. Götz, and H. Song, "Models@run.time: a guided tour of the state of the art and research challenges," *Software & Systems Modeling*, vol. 18, no. 5, pp. 3049–3082, 2019.

[14] F. Chen and G. Roşu, "Java-MOP: A monitoring oriented programming environment for Java," in *Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 546–550.

[15] H. Eichelberger and K. Schmid, "Flexible resource monitoring of Java programs," *Journal of Systems and Software*, vol. 93, pp. 163–186, 2014.

[16] T. H.-J. Uhlemann, C. Lehmann, and R. Steinhilper, "The digital twin: Realizing the cyber-physical production system for industry 4.0," *Procedia CIRP*, vol. 61, pp. 335–340, 2017.

[17] J. C. Kirchhof, J. Michael, B. Rumpe, S. Varga, and A. Wortmann, "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*, 2020, pp. 90–101.

[18] M. Vierhauser, R. Rabiser, and P. Grünbacher, "Requirements monitoring frameworks: A systematic review," *Information and Software Technology*, vol. 80, pp. 89–109, 2016.

[19] M. Mansouri-Samani and M. Sloman, "Monitoring distributed systems," *IEEE network*, vol. 7, no. 6, pp. 20–30, 1993.

[20] "Eclipse Modeling Framework (EMF)," https://www.eclipse.org/modeling/emf, last visited: 14-10-2020.

[21] Eclipse Foundation, "Xtend ," https://www.eclipse.org/xtend/, 2020, [Online; accessed 05-Oct-2020].

[22] D. Varró and A. Balogh, "The model transformation language of the VIATRA2 framework," *Science of Computer Programming*, vol. 68, no. 3, pp. 214 – 234, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016764230700127X

[23] M. Búr, G. Szilágyi, A. Vörös, and D. Varró, "Distributed graph queries over models@ run.time for runtime monitoring of cyber-physical systems," *International Journal on Software Tools for Technology Transfer*, vol. 22, no. 1, pp. 79–102, 2020.

[24] M. Vierhauser, J. Cleland-Huang, S. Bayley, T. Krismayer, R. Rabiser, and P. Grünbacher, "Monitoring CPS at runtime - A case study in the UAV domain," in *Proc. of the 44th Euromicro Conf. on Software Engineering and Advanced Applications, SEAA*, 2018, pp. 73–80. [Online]. Available: https://doi.org/10.1109/SEAA.2018.00022

[25] A. Agrawal, S. J. Abraham, B. Burger, C. Christine, L. Fraser, J. M. Hoeksema, S. Hwang, E. Travnik, S. Kumar, W. J. Scheirer, J. Cleland-Huang, M. Vierhauser, R. Bauer, and S. Cox, "The next generation of human-drone partnerships: Co-designing an emergency response system," in *Proc. of the 20th CHI Conf. on Human Factors in Computing Systems*, 2020, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3313831.3376825

[26] A. Agrawal, J. Steghöfer, and J. Cleland-Huang, "Model-driven requirements for humans-on-the-loop multi-uav missions," *Proc. of the Int'l Workshop on Model-Driven Requirements Engineering*, vol. MODRE, 2020. [Online]. Available: https://arxiv.org/abs/2009.10267

[27] I. Alexander, "Misuse cases: Use cases with hostile intent," *IEEE software*, vol. 20, no. 1, pp. 58–66, 2003.

[28] A. Gómez, J. Cabot, and M. Wimmer, "TemporalEMF: A temporal metamodeling framework," in *Proc. of the Int'l Conf. on Conceptual Modeling*. Springer, 2018, pp. 365–381.

[29] D. C. Luckham and B. Frasca, "Complex event processing in distributed systems," *Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford*, vol. 28, p. 16, 1998.

[30] M. Búr, G. Szilágyi, A. Vörös, and D. Varró, "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, 2018, pp. 111–128.

[31] A. Francalanza, J. A. Pérez, and C. Sánchez, "Runtime verification for decentralised and distributed systems," in *Lectures on Runtime Verification*. Springer, 2018, pp. 176–210.

[32] L. Baresi, L. Pasquale, and P. Spoletini, "Fuzzy goals for requirements-driven adaptation," in *Proc. of the 18th IEEE Int'l Requirements Engineering Conf.* IEEE, 2010, pp. 125–134.

[33] X. Franch, P. Grunbacher, M. Oriol, B. Burgstaller, D. Dhungana, L. López, J. Marco, and J. Pimentel, "Goal-driven adaptation of service-based systems from runtime monitoring data," in *Proc. of the 2011 IEEE 35th Annual Computer Software and Applications Conf. Workshops*. IEEE, 2011, pp. 458–463.