International Journal on Software Tools for Technology Transfer (2020) 22:79–102
https://doi.org/10.1007/s10009-019-00531-5

FASE 2018

Distributed graph queries over models@run.time for runtime monitoring of cyber-physical systems

Márton Búr[1] · Gábor Szilágyi[2] · András Vörös[2,3] · Dániel Varró[1,2,3]

Published online: 26 September 2019

Abstract
Smart cyber-physical systems (CPSs) have complex interaction with their environment which is rarely known in advance, and they heavily depend on intelligent data processing carried out over a heterogeneous and distributed computation platform with resource-constrained devices to monitor, manage and control autonomous behavior. First, we propose a *distributed runtime model* to capture the operational state and the context information of a smart CPS using directed, typed and attributed graphs as high-level knowledge representation. The runtime model is distributed among the participating nodes, and it is consistently kept up to date in a continuously evolving environment by a time-triggered model management protocol. Our runtime models offer a (domain-specific) model query and manipulation interface over the reliable communication middleware of the Data Distribution Service (DDS) standard widely used in the CPS domain. Then, we propose to carry out distributed runtime monitoring by capturing critical properties of interest in the form of graph queries, and design a distributed graph query evaluation algorithm for evaluating such graph queries over the distributed runtime model. As the key innovation, our (1) distributed runtime model extends existing publish–subscribe middleware (like DDS) used in real-time CPS applications by enabling the dynamic creation and deletion of graph nodes (without compile time limits). Moreover, (2) our distributed query evaluation extends existing graph query techniques by enabling query evaluation in a real-time, resource-constrained environment while still providing scalable performance. Our approach is illustrated, and an initial scalability evaluation is carried out on the MoDeS3 CPS demonstrator and the open Train Benchmark for graph queries.

Keywords Runtime monitoring · Graph queries · Distributed model management · Data Distribution Service (DDS)

1 Introduction

Motivation A smart and safe cyber-physical system [18,38,39,49,56] (CPS) is a software-intensive decentralized system that autonomously perceives its operational context and adapts to changes over an open, heterogeneous and distributed platform with a massive number of nodes, dynamically acquires available resources and aggregates services to make real-time decisions, and resiliently provides critical services in a trustworthy way. Several challenges of such systems have been identified in [15,17,38,39,56] including their assurance in domains like self-driving cars, autonomous drones or various Internet-of-Things applications.

Runtime models (aka models@run.time [11,57]) provide a rich knowledge representation to capture the runtime operational state and context of a smart CPS as typed and attributed graphs [21] to serve as a unifying semantic basis for runtime monitoring, management and control. On the one hand, graph models are widely used internally in various design tools for CPS (e.g., Capella, Artop). On the other hand, real-time CPSs dominantly use low-level data structures with static (i.e., compile time) memory allocation to ensure resource constraints such as memory limits or deadlines. Unfortunately, *such static data models are unable to capture dynamically*

✉ András Vörös
vori@mit.bme.hu

Márton Búr
marton.bur@mail.mcgill.ca

Dániel Varró
daniel.varro@mcgill.ca

[1] Department of Electrical and Computer Engineering, McGill University, Montreal, Canada

[2] Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary

[3] MTA-BME Lendület Cyber-Physical Systems Research Group, Budapest, Hungary

evolving contextual information where there is no a priori upper bound on relevant contextual objects (e.g., many pedestrians may be in contextual range of a self-driving car), which is a major limitation.

While runtime models can be used for different purposes, the current paper has a specific focus on their use in runtime monitoring scenarios which aims to continuously detect the violation of (safety) properties at runtime. Runtime monitoring has frequently been addressed by runtime verification (RV) techniques [40,44] which provide formal precision, but offer a low-level specification language (with simple atomic predicates to capture information about the system). Recent RV approaches [12,29] started to exploit rule-based techniques over a richer (relational or graph-based) information model. Runtime models have already been used for the assurance of self-adaptive systems (SAS) in [16,64].

Problem statement This paper aims to promote the use of graph models as a rich information source and the use of graph queries as a means of runtime monitoring in the context of distributed and resource-constrained computation platforms. However, the direct adaptation of existing graph-based techniques and tools needs to face several challenges imposed by their runtime use in a distributed smart CPS.

- **Distributed runtime models** Unlike in CPS design tools where models are stored in a centralized way and they evolve slowly, the underlying graph model needs to be distributed and updated with high frequency based on incoming sensor information and changes in network topology. In addition, we immediately need to tackle well-known challenges of distributed systems such as data consistency and fault tolerance.
- **Resource constraints and QoS requirements** In CPS design tools, models are manipulated on a single computer (or on a cloud-based server). However, graph models used at runtime need to be operated over a heterogeneous execution platform with resource-constrained devices or other quality of service (QoS) requirements (e.g., with reliable message delivery or delivery deadlines as in soft/hard real-time systems) which are frequent in edge computing or real-time embedded systems like industrial automation, telecom equipments or distributed control and simulation [50].

Existing distributed runtime models [27,28] support graph node-level versioning and reactive programming with lazy loading to make the complete virtual model accessible from every node over a Java-based platform, but not in a resource-constrained environment.

Objectives and contributions In this paper, we present a general framework for managing distributed runtime graph models with focus on runtime monitoring carried out by distributed graph query techniques specifically targeting resource-constrained smart CPSs by extending initial results presented in [12].

In particular, our contribution includes a distributed model update protocol (specified formally as statechart models), a distributed graph query evaluation algorithm and prototype implementation to provide a framework with the following key characteristics:

- **Model manipulation middleware** The runtime model offers a high-level model manipulation *interface* to be used by low-level sensors and high-level domain-specific applications, which guarantees consistent model updates in a distributed CPS setting using a novel model update *protocol*.
- **Distributed graph model with single source of truth** The graph of the runtime model is distributed over computing nodes (participants) in such a way that each graph element has a unique owner (following a known systems engineering principle).
- **Runtime monitors** Runtime monitoring is carried out by distributed graph query evaluation directly over the distributed runtime model. For that purpose, partial query results are passed to relevant participants to continue evaluation. We regard query-based monitors as a high-level protection available on top of existing (low-level) safety monitors.
- **Standard middleware for communication** Our framework uses the Data Distribution Service standard [50] as a reliable underlying messaging middleware between participants. As such, message delivery is guaranteed at a lower abstraction layer—but in a time-triggered protocol, late message delivery still needs to be handled as potential message loss.
- **Deployment to edge devices** Our graph query and manipulation middleware is thin; thus, our runtime models can be deployed over embedded or edge devices such as in the context of the MoDeS3 CPS demonstrator [65] or DDS applications.
- **Scalability** Furthermore, we carried out an initial scalability evaluation of our prototype in the context of the MoDeS3 demonstrator (as a physical CPS platform) and a simulated environment with increasing number of participants.

This paper extends our initial work [12] by providing

(i) a time-triggered distributed model update protocol which provides consistency and tolerates message losses as faults,

(ii) improvements to the distributed graph query evaluation algorithm,
(ii) deployment of our framework on top of the standard DDS communication middleware (already used in various CPS applications),
(iii) a prototype implementation and
(iv) a novel scalability evaluation which now covers both the runtime model update and query execution phase with measurements over both a physical and a simulated platform.

The rest of the paper is structured as follows. Section 2 provides an overview of our distributed model management and runtime monitoring approach. Section 3 revisits definitions related to runtime graph models. Section 4 introduces our proposed protocol for runtime model management. Section 5 revisits the foundations of local search-based graph pattern matching and describes how we applied this technique in distributed monitors. In Sect. 6, evaluation results obtained from our prototype implementation are presented. Section 7 discusses related work, while Sect. 8 with a summary and future research directions concludes our paper.

2 Overview of query-based distributed runtime monitoring

In this section, the overview of the approach is given. Figure 1 depicts the main steps and artifacts of the monitoring approach.

During the design phase, automated monitor synthesis *transforms high-level query specifications into deployable, platform-dependent source code* for each participant that will be executed as part of a monitoring service. Our approach reuses a *high-level graph query language* [61] *for specifying safety properties of runtime monitors*, which language is widely used in various design tools of CPS [55]. Graph queries can capture safety properties with rich structural dependencies between system entities which are unprecedented in most temporal logic formalisms used for runtime monitoring. Similarly, OCL has been used in [33] for related purposes. While graph queries can be extended to express temporal behavior [4], our current work is restricted to (structural) safety properties where the violation of a property is expressible by graph queries.

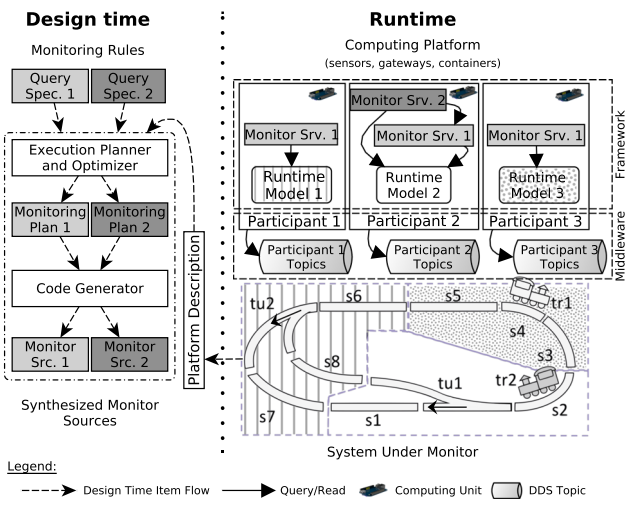


Fig. 1 Distributed runtime monitoring by graph queries

The monitor synthesis process begins with a query optimization step that transforms query specifications to platform-independent execution plans. At this point, any a priori knowledge about the runtime system provides optional input for query optimization. Execution plans are passed on to the code generator to produce platform-dependent C++ source code, which is ready to be compiled into a single executable runtime monitor program (referred to as *participant*) for the target platform. To provide better focus for the current paper, this component will not be detailed here.

Participants are *deployed to a distributed heterogeneous computation platform* which includes various types of computing units ranging from ultra-low-power microcontrollers to smart devices and high-end cloud-based servers. Some of these devices may have resource constraints (like CPU, memory).

Our system-level runtime monitoring framework is *hierarchical* and *distributed*. Monitors may observe the local runtime model of a participant, and they can collect information from runtime models of different devices, hence providing a distributed architecture. Moreover, one monitor may rely on information computed by other monitors, thus yielding a hierarchical network.

The runtime model captures data stemming from observations in the physical system. Participants are distributed across the physical system and connected via the network. These participants primarily process the data provided by their corresponding sensors, and they are able to perform edge- or cloud-based computations on the data. The runtime model management components are deployed and executed on the platform elements; thus, resource constraints need to be respected during allocation. The main responsibility of the communication middleware is to ensure timely and reliable communication between the components.

The model update and query execution messages between the components are sent over a middleware based on a publish–subscribe protocol that implements the real-time data distribution service (RDDS [35]). RDDS is an extension for the DDS standard [50] of the Object Management Group (OMG) to unify common practices concerning data-centric communication using a publish–subscribe architecture. This way, in accordance with the models@run.time paradigm [11,57], observable changes of the real system are

incorporated into the runtime model either periodically with a certain frequency or in an event-driven way upon certain triggers. Furthermore, the middleware also abstracts away the platform- and network-specific details.

Example 1 We illustrate distributed runtime models and query-based monitoring in the context of the *Model-Based Demonstrator for Smart and Safe Cyber-Physical Systems* (MoDeS3) [65], which is an educational platform of a model railway system that prevents trains from collision and derailment using safety monitors. The railway track is equipped with several sensors (cameras, shunt detectors) capable of sensing trains on a particular segment of a track connected to some participants realized by various computing units, such as Arduinos, Raspberry Pis, BeagleBone Blacks or a cloud platform. Participants also serve as actuators to stop trains on selected segments to guarantee safe operation. For space considerations, we will only present a self-contained extract from the demonstrator.

In the lower right part of Fig. 1, a snapshot of the *System Under Monitor* is depicted, where train tr1 is on segment s4, while tr2 is on s2. The railroad network has a static layout, but turnouts tu1 and tu2 can change between straight and divergent states.

Three participants are running the monitoring and controlling programs responsible for managing the different (disjoint) parts of the system marked with different patterns in Fig. 1. A participant may read its local sensors (e.g., the occupancy of a segment or the status of a turnout) and collect information from participants, and it can operate actuators accordingly (e.g., change turnout state) for the designated segment. All this information is reflected in a *distributed runtime model* which is deployed on the three computing units.

3 Preliminaries for distributed runtime models

This section revisits definitions related to metamodeling and runtime models from [12]. Additionally, we describe the required model update operations for our runtime monitoring approach, and we briefly overview the Data Distribution Service (DDS) standard [50].

3.1 Domain-specific modeling languages

Many industrial CPS modeling tools build on the concepts of domain-specific (modeling) languages (DSLs and DSMLs) where a domain is typically defined by a *metamodel* and a set of structural consistency constraints. A metamodel captures an ontology, i.e., the main concepts as classes, their attributes

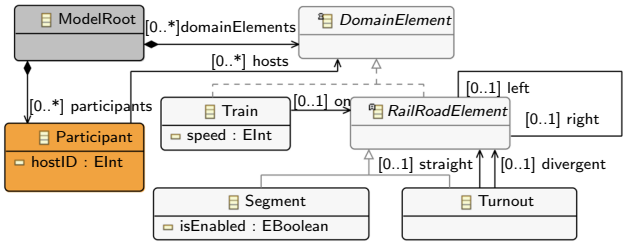


Fig. 2 Metamodel for MoDeS3

and relations as references of a domain in the form of graph models.

A metamodel can be formalized as a vocabulary $\Sigma = \{\mathtt{C}_1, \ldots, \mathtt{C}_{n_1}, \mathtt{A}_1, \ldots, \mathtt{A}_{n_2}, \mathtt{R}_1, \ldots, \mathtt{R}_{n_3}\}$ with a unary predicate symbol $\mathtt{C}_i$ for each class, a binary predicate symbol $\mathtt{A}_j$ for each attribute and a binary predicate symbol $\mathtt{R}_k$ for each relation in the metamodel.

Example 2 Figure 2 shows a metamodel for the MoDeS3 demonstrator with Participants (identified in the network by hostID attribute) which host DomainElements. A DomainElement is either a Train or RailroadElement. A Train has a speed attribute, and the train is located on a RailroadElement. Turnouts and Segments are RailroadElements with links to the left and right side RailroadElements. These left and right references describe the actual connections between the different RailroadElements. These references are navigable in both directions. Furthermore, a Turnout has additional straight and divergent references to RailroadElements to represent the possible directions.

Structural consistency constraints References in EMF metamodels involve multiplicity constraints *lo*..*up* composed of a lower bound *lo* and an upper bound *up*. In this paper, we assume that the lower bound is always 0 (which is a frequent assumption when working with incomplete models), while the upper bound can be either 1 or $*$. This way we guarantee that removing a reference will not result in a structurally inconsistent model.

We also consider bidirectional associations by adopting the concept of *opposite references* from EMF metamodels (*eOpposites*) where each reference type may have an opposite reference type and vice versa (such as left and right in Fig. 2). EMF maintains such pairs of opposite references consistently in case of non-distributed instance models, i.e., if a reference is created or deleted, its opposite reference is created or deleted automatically. However, maintaining such pairs of references is more complicated in a distributed setting, for which we are proposing a protocol in Sect. 4.2.

3.2 Runtime models

The objects, their attributes, links and dependencies between objects constitute a rich runtime knowledge base for the underlying system captured as a *runtime model* [11,57]. Relevant changes in the system or its environment are continuously reflected in an event-driven or time-triggered way in this runtime model. We assume that this runtime model is self-descriptive in the sense that it contains information about the computation platform and the allocation of various services (e.g., model management, runtime monitoring) to platform elements, which is a key enabler for self-adaptive systems [16,64].

A *runtime model* $M = \langle Dom_M, \mathcal{I}_M \rangle$ is a logic structure over Σ, as in [62], where $Dom_M = Obj_M \sqcup Data_M$, and Obj_M is a finite set of individuals (objects) in the model, while $Data_M$ is the domain of built-in data values (integers, strings, etc.).

$\mathcal{I}_M$ is an interpretation of predicate symbols in Σ defined as follows (where o_p and o_q are objects from Obj_M, and a_p is an attribute value from $Data_M$):

- **Class predicates** If object o_p is an instance of class $\mathtt{C}_i$, then the 2-valued interpretation of $\mathtt{C}_i$ in M evaluates to 1 denoted by $[\![\mathtt{C}_i(o_p)]\!]^M = 1$, and it evaluates to 0 otherwise.
- **Reference predicates** If there is a link of type $\mathtt{R}_k$ from o_p to o_q in M, then $[\![\mathtt{R}_k(o_p, o_q)]\!]^M = 1$, otherwise 0.
- **Attribute predicate** If there is an attribute of type $\mathtt{A}_j$ in o_p with value a_r in M, then $[\![\mathtt{A}_j(o_p, a_r)]\!]^M = 1$, and 0 otherwise.

3.3 Distributed runtime (graph) models

While a (regular) runtime model serves as a centralized knowledge base, this is not a realistic assumption in a distributed setting. In our distributed runtime model, each participant only has up-to-date but incomplete knowledge about the distributed system. Moreover, we assume that each model object is exclusively managed by a single participant, referred to as the *host* (i.e., *owner*) of that element, which serves as the single source of truth. This way, each participant can make calculations (e.g., evaluate a query locally) based on its own view of the system, and it is able to modify the mutable properties of its hosted model elements.

To extend the formal treatment to *distributed runtime models*, we mark which participant is responsible for storing the value of a particular predicate in its local knowledge base. For a predicate P with parameters $v_1, \ldots, v_n$, $[\![P(v_1, \ldots, v_n)]\!]^{M_d}@p$ denotes its value over the distributed runtime model M_d stored by host p.

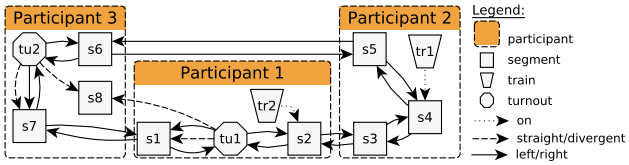


Fig. 3 Distributed runtime model for MoDeS3

Example 3 Figure 3 shows a snapshot of the distributed runtime model M_d for the MoDeS3 system depicted in the right part of Fig. 1. Participants deployed to three different physical computing units manage different parts of the system. The model represents the three participants (Participant 1–Participant 3) deployed to the computing units (depicted also in Fig. 1), the domain elements (s1–s8, tu1, tu2, tr1 and tr2) as well as the links between them. Each participant hosts model elements contained within them in the figure, e.g., Participant 2 is responsible for storing attributes and outgoing references of objects s3, s4, s5 and tr1.

3.4 Model update operations

We assume that the following model manipulation operations are available for a (distributed) runtime model:

- **Object operations** In runtime models, objects can be *created* and *deleted*. Object update operations are implemented by broadcast messages.
- **Attribute operations** Attribute values can be *updated* locally in a distributed runtime model since the values of attributes are always stored together with the object itself by the host participant.
- **Reference operations** A link can be *added* or *deleted* between two objects. If both ends of a link hosted by the same participant, then such a reference update is a local operation; otherwise, it needs to be communicated with other participants.

3.5 The Data Distribution Service middleware

The OMG specification for Data Distribution Service (DDS) [50] provides a common application-level interface for *data-centric* implementations over a *publish–subscribe* communication model. Additionally, the specification defines the main features suitable for applying in embedded self-adaptive systems. We provide a brief overview based on [50].

In data-centric systems, every data object is uniquely identified in a virtual *global data space* (shortly, GDS), regardless of its physical location. For this reason, both the applications and the communication middleware need to provide support for unique identifiers of data objects. Furthermore, this identification enables the middleware to keep only the most recent version of data upon updates, thus respecting the

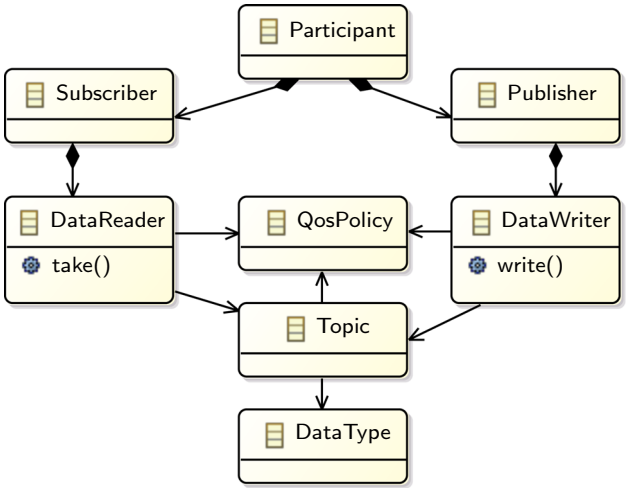


Fig. 4 UML class diagram of DDS classes

performance and fault tolerance requirements of real-time applications (that make a centralized solution impractical). By keeping the most recent data, the middleware can provide up-to-date information to new participants of the network.

A simplified metamodel capturing the concepts of DDS that implement the publish–subscribe communication model is depicted in Fig. 4. Participant is the top-level entity in a DDS application, so we assume that each deployed program has exactly one instance of it, and we refer to communicating programs as *participants*. Participants may have an arbitrary number of Subscribers and Publishers that handle the actual reading and writing of data, respectively. DataReaders and DataWriters are contained within Subscribers and Publishers. The sole role of DataWriters is to inform their corresponding Publishers that the state of the data object is changed, i.e., calling DataWriter::write() will not necessarily cause immediate communication. Similarly, the task of a Subscriber is to decide when to invoke DataReader::take() that reads the new data values.

Unlike classic publish–subscribe protocols[1,2] a Topic is more than a routing label for messages in DDS: A Topic is always associated with exactly one predefined DataType. For each DataType, a set of attributes are configured to serve as a *key*; thus, the topic and the key together are used for identifying data objects in the global data space. Additionally, this coupling between Topic and DataType (with the additional QosPolicy settings) enables implementation optimizations such as preallocating the resources needed to send or receive messages of a Topic.

Real-time DDS by [35] is an extension of the DDS standard, which tailors DDS to fit the need of real-time application scenarios. Among other novelties, the work also shows how quality of service (QoS) and quality of data (QoD) specifications can be used to ensure reliable and timely messaging, even over unstable or slow networks. Additionally, DDS is also capable of detecting and reporting violations of QoS contracts to participants. Thus, we may assume reliable and timely delivery of messages by the underlying middleware in the current work.

[1] https://mqtt.org/.

[2] https://www.amqp.org/.

4 A model management protocol for distributed runtime models

Next, a time-triggered protocol for distributed runtime model management operating over a reliable communication middleware of the DDS standard is proposed.

4.1 Overview of assumptions

Our work addresses decentralized mixed synchronous systems [58] where participants (1) communicate model updates to other participants in the first part of a time-triggered execution loop [36] (*update cycle*) and (2) then evaluate monitoring queries over a consistent snapshot of the system (*query cycle*). In [12], we focused only on the query cycle, while this paper provides a detailed description of the model update cycle as well. Below we summarize our main assumptions and considerations.

Assumptions on sensing We assume that each participant can detect relevant information about its own model elements by local sensing, which trigger model updates to its local knowledge base (together with a time stamp). The life cycle of any model element can be attached to sensor readings, i.e., creation and deletion of a train object in the runtime model depend on whether a particular sensor is able to detect the train in the real system. Such sensor readings can be periodic (e.g., once in every 10 ms) or event-driven (e.g., when a new train is detected). Raw sensor readings are buffered until the next model update cycle, while the runtime model is updated in accordance with our protocol.

Assumptions on model updates Conceptually, a participant may communicate relevant model changes to other participants either asynchronously or periodically. However, all model update requests are registered with a time stamp and buffered to be processed later in a time-triggered way by our distributed model management protocol. The real processing order of model update messages will not be time-ordered, but our protocol tolerates lost/delayed messages and handles common semantic corner cases (see later in Sect. 4.3) by the end of the model update cycle. As such, distributed graph

queries used for runtime monitoring [12] will be executed over a consistent runtime model of the system during the query cycle.

Assumptions on timeliness We assume approximate synchrony [20] between the clocks of individual computing units; thus, each participant can precisely detect the beginning and the end of each cycle wrt. each other. In other terms, the discrepancy between the clocks of participants is negligible.

The time-triggered nature of our protocol provides an upper bound defined by the cycle time of model updates (t_u) and cycle time of query evaluation (t_q). Thus, if no messages arrive late, our runtime monitors will detect a relevant situation in at most $2 * (t_u + t_q)$. However, a detailed evaluation of timeliness guarantees is out of scope for the paper and left for future work.

Assumptions on communication middleware In order to periodically communicate model changes between participants, our distributed model update protocol relies upon DDS, a standard reliable communication middleware to provide several important QoS guarantees.

1. Timely and reliable message delivery of model update messages is ensured by the DDS middleware.
2. If there is a violation of QoS guarantees, DDS notifies participants to allow them to recover from faults as part of the model update and query protocol. As such, the sender of the message will be aware of such communication fault.
3. The synchrony of physical clocks of participants is enforced by a clock synchronization protocol[41,52]; thus, each participant receives messages with a time stamp denoting when the update action was initiated.
4. Participants can save update messages to a preallocated cache with potentially limited size. This way, participants running under resource constraints will not be flooded by an excessive number of messages sent over the network, and they are able to select messages they want to keep based on their specific needs and preferences.

Assumptions on fault tolerance guarantees While DDS guarantees reliable message delivery (i.e., a sent message will eventually arrive), it may not enforce that messages would arrive within the time frame of their phase. As such, our fault model considers messages that arrive outside the time frame of their designated phase to be lost. Since DDS provides many QoS guarantees for participants, our fault model used in the paper is restricted to message loss or late arrival of a message.

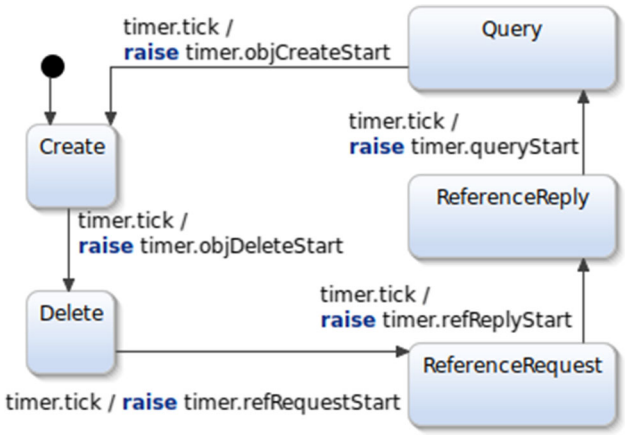


Fig. 5 Runtime phases of model updates and queries

Assumptions on properties of computing units We assume that the computing units are capable of running a program containing the implementation of the DDS standard. Based on our initial assessment to be presented in Sect. 6.3, this yields that the computing units need to have at least 15 MB heap available to successfully initialize the middleware in our case. Furthermore, the devices are assumed to be able to run the required TCP or UDP network stack.

4.2 A multi-phase model update protocol

Time-triggered execution cycle Our runtime monitoring approach is driven by a time-triggered execution loop which can be divided into five major conceptual phases. The first four phases constitute the model update protocol (discussed in this section) with (1) object create, (2) object delete, (3) link update request and (4) link update reply. The update phase is followed by a (5) query phase (discussed in Sect. 5.1).

Our model update protocol will be defined by complex statecharts containing multiple parallel regions. In this paper, we used YAKINDU Statechart Tools [67] for the specification and simulation of the protocol.

Figure 5 shows a statechart model describing this execution cycle. Transitions are triggered by events coming from a master clock that is available to all participants, which is implemented using high-precision clock synchronization across platform components.

Sensing A statechart describing the behavior of the sensing services capable of detecting objects is displayed in Fig. 6. The transitions are triggered by the events sensing.appear and sensing.disappear assumed to be raised by changes in the operational context of the system. When those transitions fire, the sensing.objectAppeared and sens-

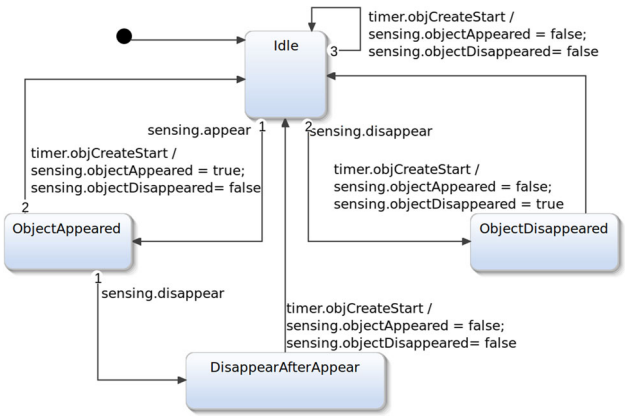


Fig. 6 High-level model of object sensing services

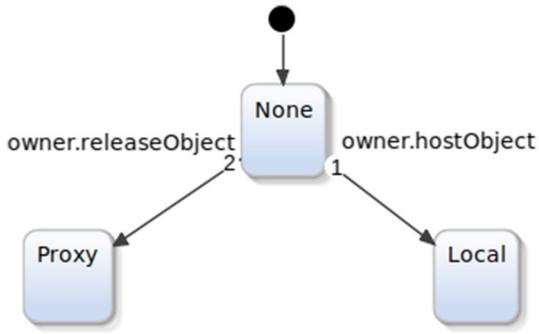


Fig. 7 Object ownership states

ing.objectDisappeared flags are set/cleared according to the events, allowing local buffering of events. These flags are cleared at the beginning of each object create cycle. Figure 6 only depicts the sensing model for objects, but an identical model employing a different pair of flags can be used to model the appearance/disappearance of links between two selected objects.

Overview of messaging for model updates In our distributed model update protocol, *object creation and deletion are communicated as broadcast messages* so that participants can register the existence of all objects. Such broadcast messages allow each participant to add or remove links to any model object, as well as to query object attribute values or links, even if an object is not hosted by the participant. On the other hand, *messages for link addition and removal are sent in a peer-to-peer manner*. The precise protocol of sending and receiving such message will be defined by a series of statecharts, which control the model update behavior individually for each model element.

4.2.1 Object create phase

The first phase of the model update cycle addresses object creation. A participant that creates a new object must send a *broadcast message* with the identifier o_{create}, the type C and its participant identifier p_{host}. Formally, the message has $[\![\mathrm{C}(o_{create})]\!]^{M_d} @ p_{host} = 1$ as content. It is necessary to notify other participants about the creation of a new object in order to allow them to create links pointing to the object (i.e., as a target end of an edge). Recipient participants will create a *proxy object* locally that represents the remote object in the model by having the same type C, but stores the object identifier o_{create} and the host participant identifier p_{host} as its only attributes. The middle column of Table 1 summarizes the actions a participant takes upon receiving an object create message.

The statechart in Fig. 8 specifies the life cycle of an object, while a parallel region (depicted in Fig. 7) shows the ownership of the object.

Statechart of object creation The initial state for an object is NoObject (Fig. 8) that represents when the object does not exist in the model, while its ownership is initially None (Fig. 7). From this initial state, creation is triggered when the participant either (1) receives a data.create message or when (2) sensing signals sensing.objectAppeared at the beginning of the object creation phase triggered by a timer.objCreateStart event.

- In the first case, the corresponding (broadcast) message arrives in the object create phase. A proxy object is registered and enters the Created state, while ownership is set to Proxy after raising event ownership.releaseObject.
- In the second case, local sensing services indicate that an object needs to be created locally by setting the sensing.objectAppeared flag and this flag is read at the start of the object create phase indicated by the timer.objCreateStart event. Then, a broadcast message is sent on the creation of a new object and the object is

Table 1 Summary of actions when receiving object update messages

	Object update message	
Condition	$[\![\mathrm{C}(obj)]\!]^{M_d} @ p = 1$ (*obj* created at *p*)	$[\![\mathrm{C}(obj)]\!]^{M_d} @ p = 0$ (*obj* deleted at *p*)
obj is unknown	Create proxy object *obj*	Create proxy for *obj* and mark it as deleted
obj is present locally	No-op	Mark proxy for *obj* as deleted

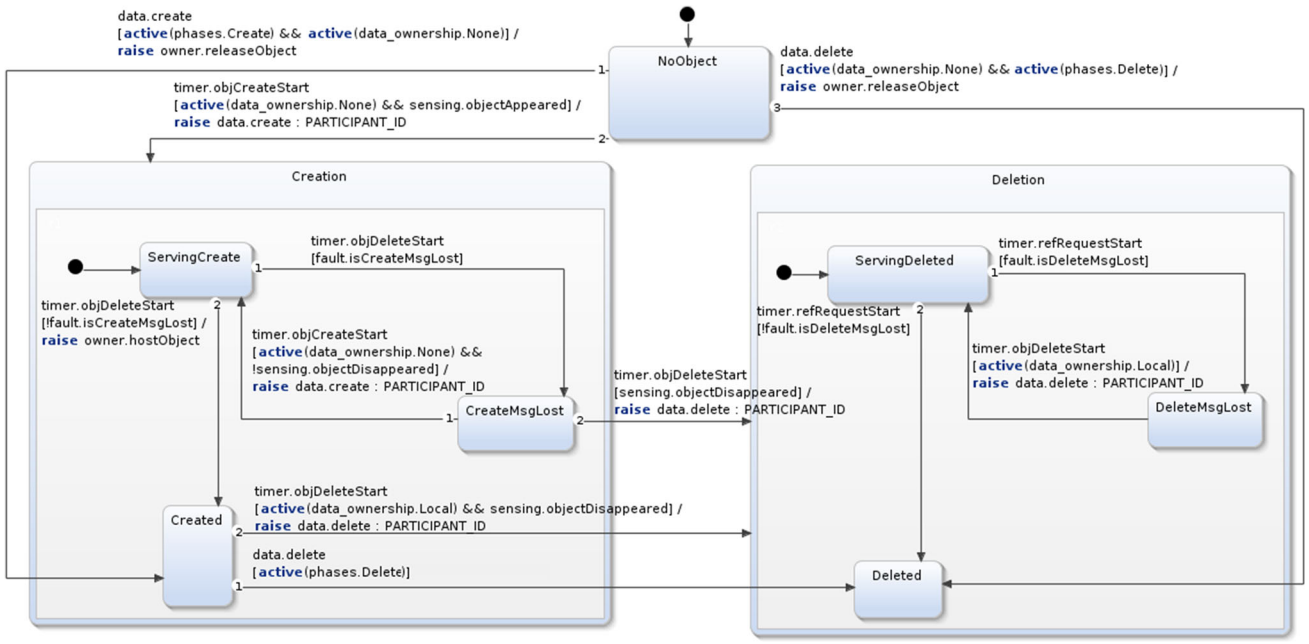


Fig. 8 Possible states of an object in our model update protocol

moved to the ServingCreate state. Once the next phase objectDelete starts, the timer.objDeleteStart event is raised, and if there was no message loss, i.e., the broadcast message was delivered to all other participants successfully, the object enters the Created state, while the ownership is set to Local.

4.2.2 Object delete phase

In the second phase of model updates, objects of the runtime model can be deleted. The phase is similar to object creation: The identifier o_{delete}, the type `C` and the host p_{host} of the deleted object are sent in a *broadcast message*. Formally, $[\![\mathtt{C}(o_{delete})]\!]^{M_d}@p_{host} = 0$ is sent, which is also saved in the local knowledge base. It is necessary to notify other participants about the deletion of an existing model element to allow them to remove potential dangling edges originally pointing to the deleted object. Only the host participant of the object can initiate the deletion of the corresponding object; otherwise, the object deletion message is ignored. Deleting an object is irreversible, i.e., once an object is deleted, it cannot be reverted. The last column of Table 1 summarizes the actions to be taken upon receiving a *delete object* message.

Statechart of object deletion Deletion of an object that is in the Created state (Fig. 8) depends on the ownership of the object (Fig. 7).

– A hosted object (i.e., ownership.Local is active) is deleted by first entering the ServingDeleted state and sending a broadcast message to all participants in the system about the deletion. Then, if all participants have been successfully notified about the deletion of the object (i.e., fault.isDeleteMsgLost is `false`) at the end of the object delete phase, the object goes to Deleted state.
– A proxy object (i.e., ownership.Proxy is active and thus owned by an other participant) is immediately brought to Deleted state upon receiving a data.delete event in the object delete phase.

Furthermore, if a participant receives a data.create for an object that is not known, i.e., while the object is in the initial state NoObject, it transitions to the Deleted state to ensure that participants have a synchronized knowledge on the state of model objects. It must be noted that an object in the Deleted state is never included in any match during the query phase.

4.2.3 Link update request phase

In this third phase of the model update protocol, link additions and removals are initiated (in arbitrary order) between objects in a peer-to-peer manner.

Link creation Adding a link from object o_{src} to o_{trg} is done without sending any messages if either (i) both objects are hosted by the same participant p_{host} or (ii) their hosts are different, but the structural consistency checks can be done locally by the host of o_{src}.

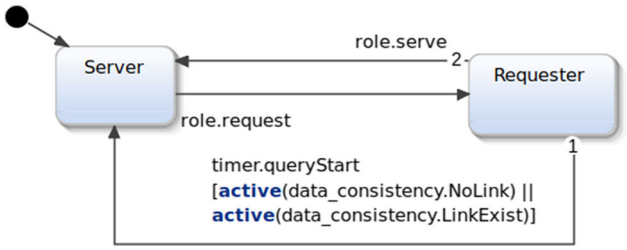


Fig. 9 Participant roles during reference update

Otherwise, link addition from object o_{src} to o_{trg} is initiated by the host of o_{src} (denoted as p_{src}). Formally, a message is sent with $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_d} @p_{src} = 1$ as content. To maintain a consistent model, the local knowledge base keeps the $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_d} @p_{src} = 0$ entry until receiving an acknowledgment message from the host of the target object containing $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_d} @p_{trg} = 1$ in the subsequent link update reply phase. However, if the link cannot be added for some reason, e.g., a multiplicity constraint would be violated, the reply from p_{trg} will be $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_d} @p_{trg} = 0$. From this information, the host of o_{src} will also deduce that the link cannot be set; thus, a consistent truth value is maintained by both parties.

Statechart of link creation Figure 10 shows the possible states of a link, while a participant's role is modeled separately in Fig. 9. The respective initial states are NoLink and Server. Similarly to object creation, adding a link can start at a participant after (1) receiving a data.addRequest while in the reference request phase or when (2) the local sensing services signal sensing.linkAppeared at the beginning of the reference request phase indicated by the timer.objCreateStart event.

- In case (1), the participant is serving the reference add request, so that it stays in the Server state, while the link transitions from NoLink to AddRequest. Then, at the start of the reference reply phase, it sends back an acknowledgment to the requester and enters the AddReply state. Upon successful delivery of the reply message, the reference is created and it enters the LinkExist state. While in AddRequest, if the new reference would violate a multiplicity constraint, a reject is sent back to the requester and the link is not created, its next state is NoLink.
- In case (2), the participant takes the Requester role and the reference moves to the AddRequest state. Once the request is successfully delivered, the reference's state changes to AddReply where it is waiting for the reply message. Once acknowledged, the link is added and it enters the LinkExist state.

Link removal The removal of a directed link leading from object o_{src} to o_{trg} is similarly done without sending any messages if either (i) both objects are hosted by the same participant p_{host} or (ii) they are hosted by different participants, but structural consistency can be ensured locally by the host of o_{src}.

Otherwise, removing a link can be initiated by participant p_{src}, the host of the source object o_{src} by sending a request to participant p_{trg} hosting the target object o_{trg}. Formally, to initiate removing a reference of type R, the content of the messages is $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_d} @p_{src} = 0$. Reference removal requests will not have corresponding reply messages, because we assumed lower multiplicity bounds for references to be 0; thus, such requests would always be acknowledged.

Statechart of link removal Table 2 briefly summarizes the actions to be taken upon receiving reference update request messages, while the right part of Fig. 10 shows the states related to the removal of an existing reference. Similarly to deleting an object, the removal of a reference that is in the state LinkExist can be triggered in two ways: either receiving a message in the reference request phase initiating the removal or via the local sensors. In the former case, there is no extra condition, the link simply goes to state NoLink. If the removal is triggered by reading sensing.linkDisappeared = `true` at the beginning of the reference request phase, the reference enters the RemoveRequested state. Once the target participant is delivered the remove message, the link goes to NoLink.

4.2.4 Link update reply phase

The only special attention is needed for handling the addition of inverse links (which need to be updated simultaneously) with [0..1] multiplicities due to the potential race condition between participants. In such a case, the target object of a link update request may reject the corresponding add request to ensure structural consistency, i.e., to respect the upper multiplicity bound. Thus, when a link with an opposite is to be added, the host of the target object needs to acknowledge the operation for the host of the source object in a subsequent *link update reply phase*.

- In case of success, both parties are consistently notified about the change by replying $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_d} @p_{trg} = 1$; thus, the opposite references can be set automatically at both participants without sending extra messages over the network.
- If a structural inconsistency is detected at the target object, the reference add request is rejected by sending $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_d} @p_{trg} = 0$.

Table 2 Summary of actions for reference update request messages

	Reference update request message	
Condition	$[\![\mathrm{R}(src, trg)]\!]^{M_d}@p = 1$ (reference add request)	$[\![\mathrm{R}(src, trg)]\!]^{M_d}@p = 0$ (reference remove request)
Link exists	No-op	Delete opposite link
Link does not exist	If multiplicity constraints hold, add opposite and send acknowledgment, otherwise send reject to request	No-op

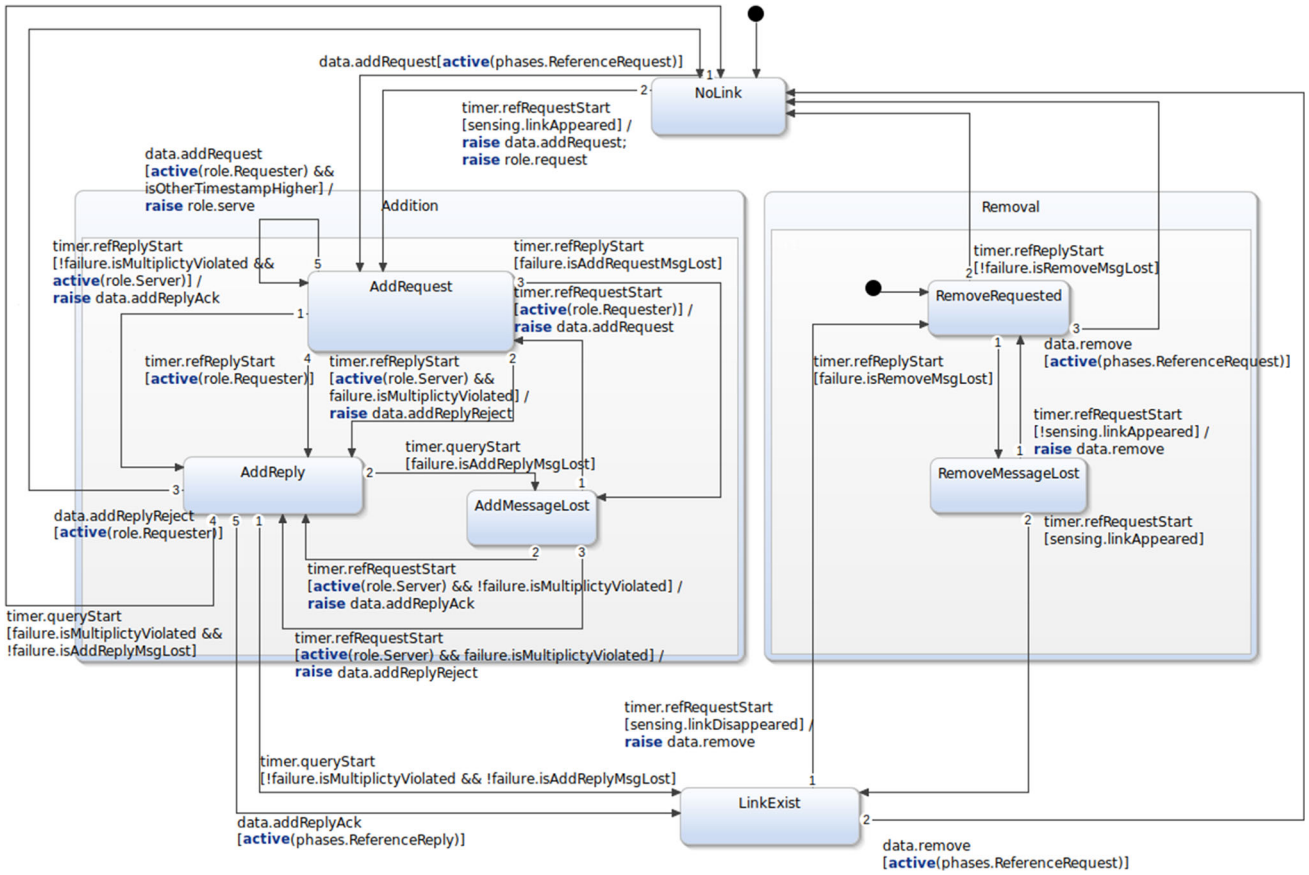


Fig. 10 Possible states of a reference in our model update protocol

4.3 Fault tolerance to handle message loss

As model update messages sent by a participant might get delayed; thus, a message will eventually arrive but possibly after its deadline (outside the respective phase). These cases are always detectable by the sender of the message, and our protocol conceptually handles such latecoming messages as message loss (i.e., the message is lost within the given cycle).

Message loss during object update Nevertheless, our object update protocol can recover from faults eventually caused by message loss thanks to extra states introduced in Fig. 8. For *object create*, if at least one message was not delivered, in ServingCreate state, fault.isCreateMsgLost is set to true based on notifications coming from the communication middleware. Then, the object enters the CreateMsgLost state and the broadcast message at the beginning of the next object create phase is repeated. This loop is iterated until eventually everyone is notified about the existence of the object.

Furthermore, the protocol is able to handle cases when a sensor reports that the object under creation should be immediately deleted (the sensing.objectDisappeared flag is set) while recovering from lost creation messages. When this happens, object enters the Deletion composite state and the deletion procedure will begin in the object delete phase.

Likewise, upon deleting an object, the DeleteMsgLost state is entered (from state ServingDelete) if the middleware detects issues with delivering the messages by the end of the object delete cycle. At the beginning of the object delete phase, the object returns to ServingDelete and the deletion broadcast messages are retransmitted. Again, this loop is iterated until eventually each remote participant is notified about the deletion.

Message loss for reference updates Similarly to object update, our reference update protocol is prepared to tolerate message loss avoid inconsistencies thanks to the extra states AddMessageLost and RemoveMessageLost introduced for fault tolerance purposes. When reference addition is requested (i.e., while having the role Server), the AddMessageLost is reached if the request message is lost. Then, at the beginning of the next reference request phase, the data.addRequest is resent and the state AddRequest is entered. If a reply message is lost as a server during a reference add, the same AddMessageLost is reached, but the reference in this case will return to the state AddReply and will retransmit the previously lost answer (either data.addReplyAck or data.addReplyReject).

Tolerating a message loss in case of a reference remove request is a simpler task compared to reference add because a remove request that is sent when transitioning from LinkExist to RemoveRequested does not need to be acknowledged. Once such a message is delivered, the edge can be safely removed until the requester is looping between RemoveMessageLost and RemvoeRequested states.

4.4 Semantic aspects of consistency

While providing a formal proof of consistency for our distributed model update protocol is outside the scope of the current paper, we highlight some aspects and corner cases which need to be tackled to establish desirable semantic properties like consistency or termination.

Termination Our protocol aims to avoid deadlocks (i.e., two participants are mutually waiting for each other) and livelocks (when they are continuously sending messages to each other). Deadlock avoidance is achieved by (1) restricting each cycle to messages of a particular type and (2) using a time-triggered execution which continuously progresses to the next phase regardless of the arrival of messages. Livelocks are avoided by ensuring that a bounded number of messages (requests and replies for each model element) are sent in each phase.

Local consistency wrt. sensor readings By local consistency, we mean that durable local events detected by sensors attached to a model element will eventually be reflected in the (local) runtime model of the participant. Since each sensor reading is recorded as a local event with a time stamp, causality of such sensing events (e.g., an object appearance or disappearance is observed by the owner participant) is easily established in the update cycle (e.g., a corresponding object is created or deleted in the runtime model), but events detected in cycle t are reflected in the runtime model in cycle $t + 1$. This gives a guarantee that the owner of a model element can make a decision based exclusively on the runtime model within at most two cycle period $2 * T$.

Global query consistency of runtime model By global consistency of the (distributed) runtime model, we mean that by the time the query cycle starts, each participant has updated its own hosted model elements, and synchronized the changes with the rest of the platform participants. As such, a query initiated by two different participants will always provide the same result set within the query cycle.

The assumed single source of truth principle (i.e., each model element has a unique owner) ensures that no contradictory updates will ever be communicated. But in case of message loss during model update phase, some participants may have outdated information about some model elements. Nevertheless, the owner of the model element will always be notified about lost messages; thus, a query accessing such a model element will still use the previous (consistent) state of the object, and the new state will be reflected when all participants are successfully notified (see below).

A potential race condition may occur when two participants attempt to add a reference between a pair of objects, but this reference also has an inverse reference with at most one multiplicity; thus, only one of the reference add operations can succeed. For a consistent model update, the one with the later time stamp should be enforced by introducing a self-loop transition in state AddRequest and one participant will act as a server, while the other will act as a requester.

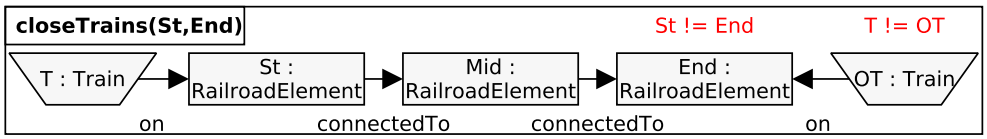
Eventual update consistency in case of message loss While global consistency prevents reading contradicting information in case of a message loss, such message loss may still prevent to delay the effects of a particular model update. In this case, according to our assumption on the communication middleware, notification is provided to the sender participant about the failure of delivering the message. This way the owner of a model element can prevent inconsistencies by tracking the last state surely known to all other participants as the consistent information and will repeatedly resend the message containing the change. For example, if some par-

```
pattern closeTrains(
  St : RailroadElement,
  End : RailroadElement
) {
  Train.on(T,St);
  Train.on(OT,End);
  T != OT;
  RailroadElement.connectedTo(St, Mid);
  RailroadElement.connectedTo(Mid, End);
  St != End;
}
```

(a) Graph query in the VIATRA Query Language

$$
\begin{aligned}
&\texttt{CloseTrains}(St, End) = \\
&\texttt{RailroadElement}(St) \wedge \\
&\texttt{RailroadElement}(End) \wedge \\
& \\
&\exists T : \texttt{Train}(T) \wedge \texttt{On}(T, St) \wedge \\
&\exists OT : \texttt{Train}(OT) \wedge \texttt{On}(OT, End) \wedge \\
&\neg(T = OT) \wedge \\
&\exists Mid : \texttt{RailroadElement}(Mid) \wedge \texttt{ConnectedTo}(St, Mid) \wedge \\
&\texttt{ConnectedTo}(Mid, End) \wedge \\
&\neg(St = End)
\end{aligned}
$$

(b) Query as formula



(c) Graphical query representation

Fig. 11 Safety monitoring objective closeTrains specified as graph pattern

ticipants are not yet notified about the creation of an object then the object is considered to be non-existing (i.e., it is in the CreateMsgLost state). This way, *update consistency is eventually achieved* when the update is successfully communicated to all recipients. Therefore, all updates will eventually take effect unless there is a more recent action which overrides its effect.

5 Distributed runtime monitoring

5.1 Graph queries for specifying safety monitors

We rely on the VIATRA Query Language (VQL) [10] to capture the properties to be monitored. VIATRA has been intensively used in various CPS design tools to provide scalable queries over large system models. Based on our previous work [12], the current paper continues to reuse this declarative graph query language for runtime monitoring. The main benefit is that monitored properties can be captured on a high level of abstraction over the runtime model, which eases the definition and comprehension of runtime monitors for engineers (compared to monitors written in an imperative language). Moreover, this kind of specification is free from any platform-specific or deployment details.

The expressiveness of the VQL language converges to first-order logic with transitive closure; thus, it provides a rich language for capturing a variety of complex structural conditions and dependencies. Technically, a graph query is used to capture an erroneous behavior, situation occurring in the runtime model. Thus, any result (or match) of a query (or pattern) highlights a violation of the safety property at runtime.

Example 4 In the railway domain, safety standards prescribe a minimum distance between trains on track [1,5]. Query closeTrains captures a (simplified) description of the minimum headway distance to identify violating situations where trains have only limited space between each other. Technically, one needs to detect whether there are two different trains on two different railroad elements, which are connected by a third railroad element. Any match of this pattern highlights track elements where passing trains need to be stopped immediately. Figure 11a shows the graph query closeTrains in a textual syntax, Fig. 11b displays it as a graph formula, and Fig. 11c shows a graphical illustration as a graph pattern.

5.1.1 Syntax

Definition 1 (*Graph query*) A graph query (or graph pattern) is a first-order logic (FOL) formula, formally $\varphi(v_1, \ldots, v_n)$, over (object and value) variables [62].

A graph query φ can be inductively constructed (see Table 3) by using atomic predicates of runtime models $\texttt{C}(v)$, $\texttt{R}(v_1, v_2)$ and $\texttt{A}(v_1, v_2)$ (with $\texttt{C}, \texttt{R}, \texttt{A} \in \Sigma$), equality between variables $v_1 = v_2$, standard FOL connectives $\vee$ and $\wedge$, quantifiers $\exists$ and $\forall$, and positive (*call*) or negative (*neg*) query calls.

The VQL language supports the hierarchical specification of runtime monitors as a query may explicitly use results of other queries (along positive or negative query calls). Furthermore, distributed evaluation will exploit a spatial hierarchy between computing units.

Table 3 Semantics of graph patterns (predicates)

1.	$[\![\mathtt{C}(v)]\!]_Z^M := \mathcal{I}_M(\mathtt{C})(Z(v))$
2.	$[\![\mathtt{A}(v_1, v_2)]\!]_Z^M := \mathcal{I}_M(\mathtt{A})(Z(v_1), Z(v_2))$
3.	$[\![\mathtt{R}(v_1, v_2)]\!]_Z^M := \mathcal{I}_M(\mathtt{R})(Z(v_1), Z(v_2))$
4.	$[\![\exists v : \varphi]\!]_Z^M := \max\{[\![\varphi]\!]_{Z, v \mapsto x}^M : x \in Obj_M\}$
5.	$[\![\forall v : \varphi]\!]_Z^M := \min\{[\![\varphi]\!]_{Z, v \mapsto x}^M : x \in Obj_M\}$
6.	$[\![v_1 = v_2]\!]_Z^M := 1$ iff $Z(v_1) = Z(v_2)$
7.	$[\![\varphi_1 \wedge \varphi_2]\!]_Z^M := \min([\![\varphi_1]\!]_Z^M, [\![\varphi_2]\!]_Z^M)$
8.	$[\![\varphi_1 \vee \varphi_2]\!]_Z^M := \max([\![\varphi_1]\!]_Z^M, [\![\varphi_2]\!]_Z^M)$
9.	$[\![\neg\varphi]\!]_Z^M := 1 - [\![\varphi]\!]_Z^M$
10.	$[\![call(\varphi(v_1, \ldots, v_n))]\!]_Z^M := \begin{cases} \exists Z' : Z \subseteq Z' \wedge \forall_{i \in 1..n} : \\ Z'(v_i^c) = Z(v_i) : [\![\varphi(v_1^c, \ldots, v_n^c)]\!]_{Z'}^M \end{cases}$
11.	$[\![neg(\varphi(v_1, \ldots, v_n))]\!]_Z^M := 1 - [\![call(\varphi(v_1, \ldots, v_n))]\!]_Z^M$

5.1.2 Semantics

Definition 2 (*Variable binding*) A variable binding Z is a mapping of query variables to objects and data values in a model M. Formally, $Z : \{v_1, \ldots, v_n\} \to Dom_M$ [62].

Definition 3 (*Bound and free variables*) A variable v_B is a bound variable in a mapping Z if v_B is mapped by Z to an element of Dom_M. Otherwise, variable v_F is a free (or unbound) variable.

A graph pattern $\varphi(v_1, \ldots, v_n)$ may be evaluated over a (centralized) runtime model M along a variable binding Z (denoted by $[\![\varphi(v_1, \ldots, v_n)]\!]_Z^M$) in accordance with the semantic rules defined in Table 3 [62]. In the rest of the paper, we may use the shorthand $[\![\varphi(v_1, \ldots, v_n)]\!]$ for $[\![\varphi(v_1, \ldots, v_n)]\!]_Z^M$ when M and Z are clear from context.

Definition 4 (*Complete match*) A variable binding Z that includes all variables of query φ is called a (complete) match if φ is evaluated to 1 over M: $[\![\varphi(v_1, \ldots, v_n)]\!]_Z^M = 1$ [62].

Definition 5 (*Partial match*) A variable binding Z_p over a subset of variables of query φ is called a partial match if all predicates in φ with bound parameters are evaluated to 1 over M.

5.1.3 Local search-based pattern matching

Graph query evaluation (aka graph pattern matching) is the process of finding all complete matches [63]. When evaluation starts, an initial binding may be given for a subset of query variables to objects and values in the model, which should be extended to a complete match.

There are multiple query evaluation strategies available [23]. Our framework uses a *local search-based pattern matching* strategy to find matches of monitoring queries based upon the foundational algorithms and model-sensitive query evaluation plans [63].

Query execution is guided by a *search plan*. When computing such search plans, the fundamental challenge is to determine the order in which all predicates included in the query should be evaluated during query execution to optimize performance. Various approaches have been proposed for this challenge [32,63], and their discussion is out of scope for this paper. In our context, search plans are regarded as an ordered list of all the predicates in a query.

During query execution, an operation is executed in each search step to evaluate the predicate corresponding to the actual step. An operation is one of the following two types based on the current binding of predicate variables to model elements:

- An **extend operation** evaluates a predicate with at least one free variable. Execution of such operations requires iterating over all potential variable substitutions and selecting the ones for which the predicate evaluates to 1.
- A **check operation** evaluates a predicate with only bound variables. Execution of such operations simply requires determining whether the predicate evaluates to 1 over the actual variable binding.

The sketch of a recursive query evaluation algorithm over a centralized model is shown in Algorithm 1. The recursive EXECUTEQUERY function takes the query φ, the index *idx* of the current operation in the search plan and a partial match Z_p as parameters. In line 2, the executor looks up the search plan for the given query from a global storage. Then, in line 3, the algorithm checks if *idx* points to the end of the operation list. If this is the case, a match has been found and should be returned. Otherwise, the matching procedure continues by initializing an empty match set (line 4) and extracting the predicate enforced in the current search step and storing this predicate to PRED (line 5). Based on the variable bindings at the current stage of the query evaluation, the algorithm

categorizes the current operation as either an extend or a check in line 6. In case of an extend operation (lines 7–11), all potential variable bindings are calculated (lines 7–8) and the predicate PRED is evaluated on them (line 9). For each new partial match Z'_p obtained this way, the matching process recursively continues with the next search step (lines 10–11). If the current search operation is categorized as a check and the execution continues in line 13, then PRED is applied over the values mapped by Z_p partial match. If this evaluation returns 1, the evaluation proceeds recursively with the next search operation (lines 14–15). Finally, in line 16, all matches found in subsequent search steps are returned. To find all matches in the model for query φ, EXECUTEQUERY should be called with (φ, 0, Z_p) parameters (lines 17–18).

Algorithm 1 Query execution algorithm outline

```
1: function ExecuteQuery(φ, idx, Z_p)
2:     searchPlan ← LookupPlan(φ)
3:     if size(searchPlan) = idx then return {Z_p}
4:     matches ← ∅
5:     PRED ← predicate evaluated by searchPlan[idx]
6:     if searchPlan[idx] is an extend then
7:         for e in {all candidates in M} do
8:             Z'_p ← Z_p ∪ {v_F ↦ e}
9:             if [[PRED]]^M_{Z'_p} = 1 then
10:                next ← idx + 1
11:                matches ← matches ∪ ExecuteQuery(φ, next, Z_p)
12:    else
13:        if [[PRED]]^M_{Z_p} = 1 then
14:            next ← idx + 1
15:            matches ← matches ∪ ExecuteQuery(φ, next, Z_p)
16:    return matches
17: procedure FindAllMatches
18:    allMatches ← ExecuteQuery(φ, 0, ∅)
```

Example 5 Table 4 shows a possible search plan for the closeTrains query. Each row represents a search operation. The first column is the assigned operation number (or index). The second column (predicate) shows which predicate is evaluated by the given step, and the third column shows the variables that are already bound by the previous operations when the current operation begins execution. The fourth column shows the search operation type (check or extend) which is based on the variable bindings prior to the execution of the search operation: If the predicate parameters are all bound, then it is a check; otherwise, it is an extend.

5.2 Execution of distributed runtime monitors

To evaluate graph queries in a distributed setting, we propose to deploy queries to the same target platform where the runtime model is maintained in a way that is compliant with the distributed runtime model and the potential resource restrictions of computing units in the platform. If a graph query engine is provided as a service by a participant, it can serve as a *local monitor* over the runtime model. However, such local monitors are usable only when all objects of the graph traversed and retrieved during query evaluation are stored by the same participant, which is not the general case. Therefore, while the local evaluation of queries is still preferable for performance reasons, a *distributed monitor* needs to gather information from remote model fragments constituting the distributed model M_d and monitors run by different participants.

5.2.1 A query cycle

Monitoring queries are evaluated during the so-called *query cycle*. We assume that the search plan for each monitoring query has been made available to all participants prior to this query cycle phase and each participant uses this same search plan for query evaluation.

When participants compute matches in a distributed way, they simultaneously evaluate predicates of the query on the values of the bounded variables. However, in cases when a predicate evaluation cannot be computed based on the local knowledge of a participant, the matching should be delegated to the participant hosting the corresponding part of the distributed runtime model M_d. The delegation is possible through proxies representing remote objects in the local runtime model.

We extend Algorithm 1 for a distributed platform shown in Algorithm 2. A monitor execution at a participant can be initiated by calling FINDALLMATCHES (line 24 in Algorithm 2). There were two key cases that had to be taken into consideration:

- *Delegating execution* The distributed runtime model M_d refers to the unified knowledge of multiple participants about the system, where each element of the model is owned by a single participant. This way, if the distributed query execution algorithm is finding matches over the complete runtime model, it needs to take into account matches formed by joining the locally stored parts of the complete model. To support the distributed execution, we added an extra condition for evaluating extend search operations to check whether the value for the newly bound variable is part of the local knowledge base. If this is not the case, query execution is delegated to the owner of the data. This extension is shown in lines 12–13 in Algorithm 2, while receiver will execute the CONTINUE procedure.
- *Gathering matches* Delegating a query execution to a remote participant can be done asynchronously in accordance with the actor model [31] (see lines 13–14 in Algorithm 2). This way, finding local matches can

Table 4 Search plan for query closeTrains

#idx	Predicate	Bound variables	Type
0	`Train`(T)		Extend
1	`On`(T, St)	T	Extend
2	`RailroadElement`(St)	T, St	Check
3	`ConnectedTo`(St, Mid)	T, St	Extend
4	`RailroadElement`(Mid)	T, St, Mid	Check
5	`ConnectedTo`(Mid, End)	T, St, Mid	Extend
6	`RailroadElement`(End)	T, St, Mid, End	Check
7	$\neg(St = End)$	T, St, Mid, End	Check
8	`On`(OT, End)	T, St, Mid, End	Extend
9	`Train`(OT)	T, St, Mid, End, OT	Check
10	$\neg(T = OT)$	T, St, Mid, End, OT	Check

continue without waiting for replies from remote participants. However, the execution finishes, i.e., neither FINDALLMATCHES nor CONTINUE cannot be completed before awaiting all matches from remote participants and fusing them with the local results (see lines 27–28 and lines 22–24, respectively).

Algorithm 2 Distributed query execution outline

```
1: function ExecuteQuery(φ,idx,Z_p)
2:     searchPlan ← LookupPlan(φ)
3:     if size(searchPlan) = idx then return {Z_p}
4:     matches ← ∅
5:     PRED ← predicate evaluated by searchPlan[idx]
6:     if searchPlan[idx] is an extend then
7:         for e in {all candidates in M} do
8:             Z'_p ← Z_p ∪ {v_F ↦ e}
9:             if [[PRED]]^M_{Z'_p} = 1 then
10:                next ← idx + 1
11:                matches←matches∪ExecuteQuery(φ,next,Z_p)
12:            if e is not owned by current participant then
13:                future←Continue(sender,φ,idx,Z'_p)
14:                store future
15:    else
16:        if [[PRED]]^M_{Z_p} = 1 then
17:            next ← idx + 1
18:            matches←matches∪ExecuteQuery(φ,next,Z_p)
19:    return matches
20: procedure Continue(sender,φ,idx,Z_p)
21:    matches←ExecuteQuery(φ, idx, Z_p)
22:    await all futures stored in ExecuteQuery
23:    add remote results to matches
24:    send matches to sender
25: procedure FindAllMatches
26:    allMatches ← ExecuteQuery(φ, 0, ∅)
27:    await all futures stored in ExecuteQuery
28:    add remote results to allMatches
```

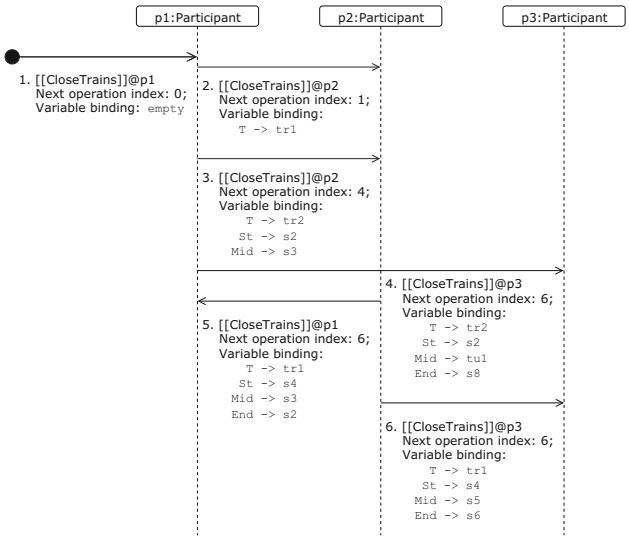


Fig. 12 Query execution requests across participants while evaluating closeTrains

Example 6 Figure 12 shows the beginning of a query evaluation sequence for monitor closeTrains initiated at Participant 1 over the runtime model depicted in Fig. 3. Calls are asynchronous (cf. actor model),

and numbers represent the order between time stamps of messages. In this example, only (the first few) requests are shown and replies are omitted to keep the illustrative example simple.

When the query is initiated (message 1, shortly, m1), its query identifier is sent along with the initially empty list of variable bindings and the search operation index 0. Then, according to the first search operation, participant p1 will look for appropriate variable values variable T potentially satisfying the predicate `Train`. Two objects are considered: tr1 and tr2 out of which tr1 is managed by the remote participant p2, so that m2 is sent from p1 to p2 delegating query

execution by supplying the query identifier, the index of the next search operation and the computed variable binding. Once p1 sent the message in a non-blocking way, it proceeds with the execution of the search plan. Next, when the binding with T → tr2, St → s2 and Mid → s3 is computed by p1 in search step #3, it is sent to p2 in m3 along with the next operation index, 4. The next message m4 sent by p1 is when it computes the T → tr2, St → s2, Mid → tu1 and End → s8 binding and sends it to p3, the host of s8. The last two messages depicted in Fig. 12, m5 and m6, are sent by p2 and are both follow-up messages for the initial request sent by p1 in m2.

5.2.2 Semantics of distributed query evaluation

Each query is initiated at a designated computing unit which is responsible for calculating query results by aggregating the results retrieved from its neighbors. This aggregation consists of computing matches based on the local knowledge base and waiting for matches from remote participants and adding them to the result set.

Message delays and/or message losses may cause participants to wait for results infinitely. This issue is handled by the time-triggered execution, where the end of the query cycle will force participants to report their results. As such, any unfinished partial matches can be reported as potential violations of a safety property, which may result in false positive alerts, but critical situations will not be missed.

5.2.3 Performance optimizations

Each match sent as a reply to a participant during distributed query evaluation can be cached locally to speed up the re-evaluation of the same query within the query cycle. This *caching of query results* is analogous to *memoing* in logic programming [66].

Currently, cache invalidation is triggered at the end of each query cycle by the local physical clock, which we assume to be (quasi-)synchronous with high precision across the platform.

This memoing approach also enables units to selectively store messages in the local cache depending on their specific needs. Furthermore, this can incorporate to deploy query services to computing units with limited amount of memory and prevent memory overflow due to the several messages sent over the network.

5.2.4 Semantic guarantees and limitations

Consistency Our approach ensures that query execution initiated at any two participants will not yield contradicting query results. This is achieved by the single source of truth principle when only an owner of an object can serve a read request during query execution.

Furthermore, in case of a communication failure, the results may contain incomplete or uncertain matches by the end of the query cycle. However, (1) these will overestimate the complete set of query results and (2) two result sets obtained by two different platform units will still not contradict each other.

Termination We can guarantee that query evaluation terminates despite potential message losses, i.e., there is no deadlock or livelock in the distributed query protocol. To show this property, it is enough to see that (1) the evaluation of queries is a monotonous process in terms of search operation execution and (2) a search operation cannot halt the execution. Condition (1) holds because whenever a participant is executing an operation that incurs query delegation, the delegation will start from the next operation in the plan. This way execution will never go back to a previous operation. Condition (2) holds because the model size is bounded; thus, all model elements can be traversed.

Assumptions and limitations There are also several assumptions and limitations of our approach. We only assumed delay/loss of messages, but not the failures of computing units. We also excluded the case when participants maliciously send false information. Instead of refreshing local caches in each cycle, the runtime model could incorporate information aging which may enable to handle other sources of uncertainty (which is currently limited to consequences of message loss). Finally, in case of longer cycles, the runtime model may no longer provide up-to-date information at query evaluation time. We believe that some of these limitations can be handled in future work by various adaptations of the query evaluation protocol.

6 Evaluation

We conducted measurements to evaluate the scalability of our distributed runtime model and query evaluation technique to address the following research questions:

Q1: *How does the distributed model update technique scale with increasing size of models and number of participants?*

Q2: *How does the distributed graph query execution technique perform with increasing model size and number of participants?*

As a main conceptual extension wrt. [12], we conducted measurements for (1) both model update and query phases (2)

with increasing number of participants, (3) on both physical and virtual platforms (4) running DDS as communication middleware.

6.1 Measurement setup

We carried out experiments on two different platforms to increase the representativeness of our measurements.

Real CPS platform We used the real distributed (physical) platform of the CPS demonstrator which consists of 6 interconnected BeagleBone Black (BBB) devices (all running embedded Debian Jessie with PREEMPT-RT patch) connected to the railway track itself. This arrangement represents a distributed CPS with several computing units having only limited computation and communication resources. We used these units to maintain the distributed runtime model, and evaluate monitoring queries. This way we are able to provide a realistic evaluation; however, we cannot evaluate the scalability of the approach wrt. the number of computing units due to the fixed number of devices in the platform.

Real CPS benchmark For the measurements over the real CPS platform, we rely on the MoDeS3 railway CPS demonstrator as the domain of our experiments to synthesize various distributed runtime models. Since the original runtime model of MoDeS3 has only a total of less than 100 objects and a total of six participants, we scaled up this initial model. To ensure that structurally consistent models are generated, we followed a template-based method, which is a simplified version of [30]. Altogether, we used the same model generator and queries as in [12] to obtain comparable results.

Virtual CPS platform To evaluate scalability wrt. increasing number of participants, we deployed our framework over a virtual platform with Docker containers. This way, we can increase both the model size and dynamically add new participants. The containers were running Ubuntu Linux 18.04 LTS, and they were all deployed to the same server machine with 32 cores and 240 GB memory. A dedicated Docker network was created and assigned to the containers allowing them to communicate over a virtual local area network.

Virtual CPS benchmark For the measurements over the virtual platform, we used the model generator and graph queries of the open Train Benchmark [54] by making only the necessary technological adaptations for a DDS-compatible execution platform.

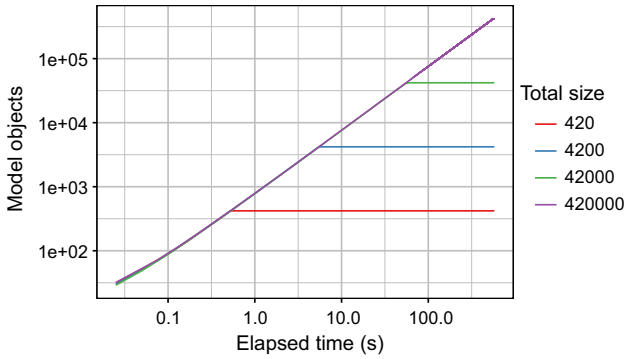


Fig. 13 Number of model objects registered by a single participant

DDS middleware We used a commercial DDS implementation provided by RTI[3] which supports the QoS settings included in the DDS specification. Furthermore, RTI provides additional options to fine-tune applications. We made minor modifications to the initial profile provided in *high_throughput.xml* to ensure timely message delivery. Namely, we increased the *max_samples* for the data writer to allow increased write throughput. Furthermore, we set the `max_flush_delay` to 100 ms to ensure periodic sending of buffered messages and increased the `max_send_window_size` to allow larger batches of transport messages. These two parameters are both RTI's own extensions to the standard.

6.2 Benchmark results over real CPS platform

6.2.1 Model update throughput

In the first set of experiments, we assessed how the model update throughput is affected by the size of the runtime model. Each BBB was running a single participant, while each participant was sending 70, 700, 7000 and 70,000 broadcast update messages, while also listening to model updates sent by other participants.

Figure 13 shows our results. Each line represents a separate scenario where 420, 4.2k, 42k and 420k objects in total were created by the participants, respectively. Furthermore, lines in the plot depict the median of how many objects a *single participant* registered over time during the experiment (both local and remote objects).

Figure 13 implies that the throughput of model updates is not affected by the actual size of the model or the number of participants. The average throughput measure is processing 797 object updates per second. Additionally, the results also point out that our approach *scales up to 420k model objects hosted across 6 participants*.

[3] https://www.rti.com/products/dds-standard.

Table 5 Memory footprint of the prototype

Objects (count)	Total memory (MB)	Model size (MB)	Avg. object footprint (Byte)
420	14.77	0.59	1404.76
4,200	15.89	1.71	407.14
42,000	27.76	13.58	323.33
420,000	146.50	132.32	315.05

Conclusion The evaluation results show that the performance of the object update protocol is independent from the number of model objects already present in the model, i.e., the creation/update of one object does not depend on the size of the entire model, which is a key property for scalability.

6.3 Memory footprint

We measured the heap memory consumption of our prototype maintaining a runtime model on a single BeagleBone unit. As a baseline, we measured the total memory consumption (second column in Table 5) including memory allocated for the required DDS data structures without creating any model objects. Then, we created 420, …, 420k model objects with their references and checked the total memory consumption (third column in Table 5). Based on this, we calculated the average memory consumption of an object (fourth column in Table 5).

Conclusion Concerning the memory use of a single runtime model object (approx. 300–400 bytes for larger models), we consider our runtime model to be lightweight, which is very promising in terms of scalability wrt. model size.

Limitation However, the loaded libraries and initial DDS data structures (mainly DDS topics) in our setup prevent our prototype to be deployed on devices with less than 15 MB memory. Note that only around 3 MB of memory is dedicated to message buffers introduced by our middleware configuration (i.e., to send batch messages to increase throughput), and the rest of memory consumption would be noticed for any DDS-based implementation using the same (industrial) library. The measured memory usage is in accordance with the memory benchmark results provided by RTI.[4]

In fact, there is a new standard called DDS-XRCE [26] dedicated to low-memory devices. This standard is an extension of the initial DDS specification designed to support resource-constrained environments, which could provide a much lower runtime overhead. However, as of today, no implementation was available to us.

[4] https://www.rti.com/products/benchmarks.

6.3.1 Query execution times

The query execution times over models deployed to a single BBB were first measured to obtain a *baseline evaluation time of monitoring* for each rule (referred to as *local* evaluation). Then the execution times of system-level distributed queries were measured over the platform with 6 BBBs, evaluating two different allocations of objects (*standard* and *alternative* evaluations).

In Fig. 14, each result captures the times of 30 consecutive evaluations of queries. A query execution starts when a participant initiates evaluation, and terminates when all participants have finished collecting matches and sent back their results to the initiator.

Overhead of distributed evaluation On the positive side, the performance of graph query evaluation on a single unit is comparable to other graph query techniques reported in [54] for models with approximately 0.5 M objects, which shows a certain level of maturity of our prototype. Furthermore, the CPS demonstrator showed that distributed query evaluation yielded runtimes, which are comparable with runtimes yielded by local evaluation on models over 4200 objects. However, distributed query evaluation was the slowest on larger models and had problems with *Train locations*, which is a simple query with large result set size that roughly equals to 10% of the complete model size; thus, communication of results imposes intense network traffic.

Altogether, our measurement results in Fig. 14 indicate one order of magnitude better scalability for query execution compared to results reported in [12].

Impact of allocation on query evaluation Similarly as in [12], we synthesized different allocations of model elements to computing units to investigate the impact of allocation of model objects on query evaluation. With the real CPS benchmark model in particular, we chose to allocate all Trains to a dedicated BBB, and assigned every other node stored previously on this BBB to the rest of the participants.

Interestingly, the difference in runtimes compared to the standard distributed scenario is negligible, while previous results [12] showed 19.92× slowdown for extreme cases. However, since that initial prototype, we managed to signif-

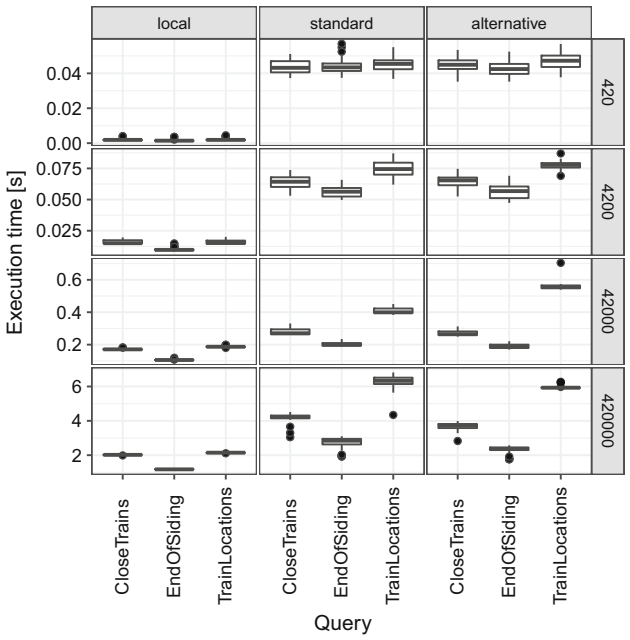


Fig. 14 Query execution times in MoDeS3 case study

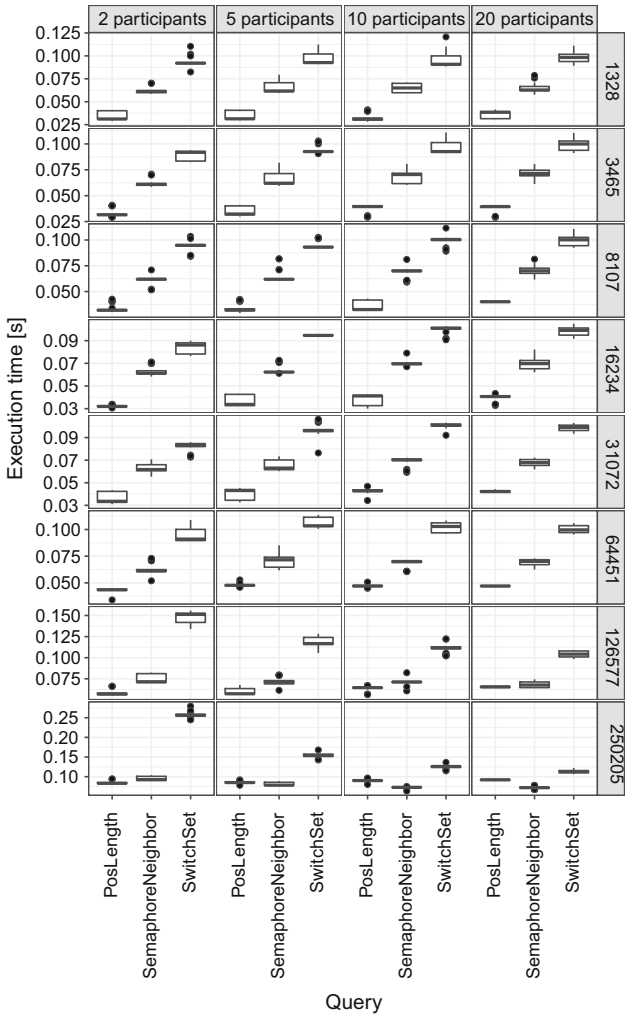


Fig. 15 Train Benchmark scalability evaluation results

icantly improve the distributed query evaluation algorithm, and exploit the high data throughput of the DDS communication middleware.

Conclusion As a combined effect of our optimized distributed query evaluation algorithm and the use of DDS as the underlying communication middleware, scalability of distributed query evaluation improved significantly, i.e., we can evaluate queries over runtime models which are an order of magnitude larger than the ones presented in [12].

Threats to validity The generalizability of our experimental results is limited by certain factors. First, to measure the performance of our approach, the platform devices (1) executed only query services and (2) connected to an isolated local area network via Ethernet. Performance on a real network with a busy channel would likely have longer delays and message losses, thus increasing execution time. Then we assessed performance using a single query plan synthesized automatically by the VIATRA framework but using heuristics to be deployed for a single computation unit. We believe that execution times of distributed queries could further decrease with a carefully constructed search plan.

6.4 Virtual CPS benchmark results

Scalability of query evaluation over a virtual platform With the virtual CPS platform, we aimed at assessing how our query-based runtime monitoring approach performs wrt. the number of participants in the platform. To achieve this, we adapted the model generator component of Train Benchmark [54] to also supply the generated models with allocation information. Then, we generated models with objects 1.3–250k for four different allocations to 2, 5, 10 and 20 participants, respectively. Figure 15 shows the runtimes of 30 subsequent query evaluations over a virtual CPS platform consisting of multiple Docker containers.

The results show that initially, query execution times are approximately the same for all allocations. Then, starting from 64k objects, execution times over the same model size gradually start decreasing as the number of participants increases. The biggest gain on average is for the query SwitchSet query: Evaluation on a platform with 20 participants over a model with 250k elements is 2.28× faster than on a platform with only 2 participants. This means that increasing the degree of distribution in the system yields lower execution times for queries if the models are larger than a certain size.

Conclusion Increasing the number of participants results in lower query evaluation time for models of a given size, but there is room for improvement for the actual speedup factor.

Threats to validity. The generalizability of our experimental results is limited by certain factors. First and foremost, we ran Linux containers on a remote server located in a cloud infrastructure, so that we had very limited influence on the allocation of the machines and the potential workload that is present on the same physical host as our instances. Furthermore, the network is also virtualized and the latency of a virtualized network is most likely significantly lower than a physical one. Additionally, to measure the performance of our distributed model management approach, participants executed only model management tasks. Finally, we did not use fault injection to investigate the fault tolerance capabilities of our protocol by enforcing message losses.

7 Related work

Runtime models The models@run.time paradigm [11] serves as the conceptual basis for the Kevoree framework [46]. This framework originally aimed at providing an implementation and adaptation of the de facto EMF standard for runtime models [22]. KMF allows sharing objects between different *nodes*, as opposed to our current work where the model elements can only be modified by their host participant, thanks to the single source of truth principle. Additionally, several assumptions applied to KMF heavily depend on the Java programming language and the Eclipse modeling framework, which questions its applicability to resource-constrained environments.

The work presented in [28] combines reactive programming, peer-to-peer distribution and large-scale models@run.time to leverage the challenges introduced by constantly changing runtime models. The basic idea is to asynchronously communicate model changes as chunks, where chunks can be processed individually regardless of other elements in the model. A prototype for this approach is also provided as an extension to KMF.

Other recent distributed, data-driven solutions include the global data plan (GDP) [68]. This work suggests a data-centric approach for model-based IoT systems engineering with a special focus on cloud-based architectures, providing flexibility and access control in terms of platform components and data produced by sensors. However, data are represented by time series logs, which is considered as low-level representation compared to graph models employed by our approach.

Another distributed solution for the models@run.time is presented in [64] as a modeling language for executable runtime megamodels (EUREMA). This project is primarily concerned with specifying the self-adaptation strategy following a model-based approach—while the data storage and representation is out of its scope.

Adaptive exchange of distributed partial models was studied in [25]. The authors propose a role-based model synchronization approach for efficient knowledge sharing. First, they identify three strategies for model synchronization. Then, with the help of different roles, they show optimizations for knowledge sharing in terms of performance, energy consumption, memory consumption and data privacy. In contrast, data ownership is exclusive, and based on the platform in our approach and global approach, system-level queries are computed based on the local information.

Distributed graph databases There are existing databases that use graphs as the underlying data representation. One of such databases is JanusGraph (formerly known as TITAN) [59]. It provides support for storing and querying very large graphs by running over a cluster of computers. In addition to storing data in a distributed way within a cluster, it also supports fault tolerance by replication and multiple simultaneous query executions by transactions. Even though it claims to execute complex graph traversals in real time, the framework provides no QoS assurance regarding response time.

OrientDB [14] is a multimodel database that has a native graph database engine where graph data may or may not be defined by a corresponding schema. However, in case of both JanusGraph and OrientDB, deployment of the database to memory-constrained devices is not supported by default, which is a fundamental need for distributed CPSs.

The authors in [27] introduce GreyCat, an implementation for *temporal graphs*. By adding time stamps to graph nodes, it allows identifying a node along its timeline. The tool can be used on top of arbitrary storage technologies, such as in-memory or NoSQL databases. As opposed to our approach, they use a per-node locking approach to prevent inconsistencies.

Finally, it is worth pointing out that the adaptation of traditional design time modeling approaches from model-driven software engineering to runtime models introduced in this current paper also fits the general research directions suggested in [7], while in [42] DDS is suggested as a key enabler technology for allowing timely and reliable data delivery for modern model-based applications.

Runtime verification approaches For continuously evolving and dynamic CPSs, an upfront design time formal analysis needs to incorporate and check the robustness of component behavior in a wide range of contexts and families of

configurations, which is a very complex challenge. Thus, consistent system behavior is frequently ensured by runtime verification (RV) [40], which checks (potentially incomplete) execution traces against formal specifications by synthesizing verified runtime monitors from provenly correct design models [34,44]. These approaches focus on the temporal behavior of the system: Runtime verification of data-driven behavior is not their main goal.

Recent advances in RV (such as MOP [43] or LogFire [29]) promote to capture specifications by rich logic over quantified and parameterized events (e.g., quantified event automata [8] and their extensions [19]). Moreover, Havelund proposed to check such specifications on-the-fly by exploiting rule-based systems based on the RETE algorithm [29]. However, these techniques consider relations between events and do not take models as a first class citizen of the runtime analysis.

Traditional RV approaches use variants of temporal logics to capture the requirements [9]. Recently, novel combinations of temporal logics with context-aware behavior description [2,24] (developed within the R3-COP and R5-COP FP7 projects) for the runtime verification of autonomous CPS have appeared and provided a rich language to define correctness properties of evolving systems. These approaches introduced the concept of context models and can also be represented in the graph-based approach of this paper. Recently, monitoring approaches to analyze topological properties in a discrete space have appeared [48]. Qualitative and quantitative analysis is supported. However, complex data-driven behavior is not the focus of the approach.

Runtime verification of distributed systems While there are several existing techniques for runtime verification of sequential programs, the authors of [47] claim that much less research was done in the area for distributed systems. Furthermore, they provide the first sound and complete algorithm for runtime monitoring of distributed systems based on the 3-valued semantics of LTL.

The recently introduced Brace framework [69] supports RV in distributed resource-constrained environments by incorporating dedicated units in the system to support global evaluation of monitoring goals. There is also focus on evaluating LTL formulae in a fully distributed manner in [3] for components communicating on a synchronous bus in a real-time system. These results can provide a natural extension of our work into the temporal directions. Additionally, machine learning-based solution for scalable fault detection and diagnosis system is presented in [6] that builds on correlation between observable system properties.

Distributed graph queries Highly efficient techniques for local search-based [13] and incremental graph model queries [60] as part of the VIATRA framework were developed, which mainly builds on RETE networks as a baseline technology. In [53], a distributed incremental graph query layer deployed over a cloud infrastructure with numerous optimizations was developed. Distributed graph query evaluation techniques were reported in [37,45,51], but none of these techniques considered an execution environment with limited resources.

8 Conclusions and future work

In this paper, we proposed a runtime verification approach for distributed CPS. The solution is a time-triggered and distributed runtime model management approach that keeps the information in a model close to the data sources. Models and high-level graph queries provide an expressive language to capture correctness requirements during runtime. Our solution is built on top of the standard DDS reliable communication middleware that is widely used in self-adaptive and resource-constrained CPS applications. The main goal of this paper was to extend our former solution [12] with the efficiency and QoS guarantees provided by DDS.

Our approach introduces an efficient handling of a distributed knowledge base stored as a graph over a heterogeneous computing platform. Consistent manipulation and update of the knowledge base are defined as a distributed and time-triggered model management protocol and implemented by exploiting the QoS guarantees provided by the DDS communication middleware.

The scalability of our approach was evaluated in the context of the physical system of MoDeS3 CPS demonstrator with promising results such as high throughput for model updates and good scalability with increasing change sizes and number of participants.

In the future, we will investigate in details what general properties does the proposed distributed runtime model protocol guarantee (e.g., global consistency, fairness and liveness). The results will be formulated as theorems with formal proofs.

Moreover, as a long-term goal, we plan to integrate the graph query-based approach with temporal logic languages to support an even wider range of specifications. In addition, more efficient query evaluation algorithms have to be incorporated into the system to provide near real-time analysis capabilities.

Acknowledgements Open access funding provided by Budapest University of Technology and Economics (BME). We are grateful for Gábor Szárnyas for the help with running Train Benchmark, Zsolt Mázló for the help with preparing the MoDeS3 platform for evaluation, Kristóf Marussy for the insightful comments and the helpful feedback received

from the anonymous reviewers. This paper is supported by the NSERC RGPIN-04573-16 project, the McGill Engineering Doctoral Award (MEDA), the MTA-BME Lendület Cyber-Physical Systems Research Group and the ÚNKP-17-2-I New National Excellence Program of the Ministry of Human Capacities. The research has been supported by the European Union and co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications) and by the National Research, Development and Innovation Fund (Thematic Excellence Program, TUDFO/51757/2019-ITM).

References

1. Headways on high speed lines. In: 9th World Congress on Railway Research, pp. 22–26 (2011)
2. Scenario-based automated evaluation of test traces of autonomous systems. In: DECS workshop at SAFECOMP (2013)
3. Decentralised LTL monitoring: Formal Methods in System Design **48**(1–2), 46–93 (2016)
4. Foundations for streaming model transformations by complex event processing. Software & Systems Modeling, pp. 1–28 (2016)
5. Abril, M., Barber, F., Ingolotti, L., Salido, M., Tormos, P., Lova, A.: An assessment of railway capacity. Transp. Res. Part E Logist. Transp. Rev. **44**(5), 774–806 (2008)
6. Alippi, C., Ntalampiras, S., Roveri, M.: Model-free fault detection and isolation in large-scale cyber-physical systems. IEEE Trans. Emereg. Topics Comput. Intell. **1**(1), 61–71 (2017)
7. Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: FoSER (2010)
8. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard D.E.: Quantified event automata: towards expressive and efficient runtime monitors. In: FM, pp. 68–84 (2012)
9. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14 (2011)
10. Bergmann, G., Ujhelyi, Z., Ráth, I., Varró, D.: A graph query language for EMF models. In: Theory and Practice of Model Transformations—4th International Conference, ICMT 2011, Zurich, Switzerland, June 27–28, 2011. Proceedings, pp. 167–182 (2011)
11. Blair, G.S., Bencomo, N., France, R.B.: Models@run.time. IEEE Comput. **42**(10), 22–27 (2009)
12. Búr, M., Szilágyi, G., Vörös, A., Varró, D.: Distributed graph queries for runtime monitoring of cyber-physical systems. In: 21st International Conference on Fundamental Approaches to Software Engineering (FASE), pp. 111–128 (2018)
13. Búr, M., Ujhelyi, Z., Horváth, Á., Varró, D.: Local search-based pattern matching features in EMF-IncQuery. In: ICGT, Vol. 9151, pp. 275–282. Springer (2015)
14. CallidusCloud. Orientdb (2018)
15. Cengarle, M., Bensalem, S., McDermid, J., Passerone, R., Sangiovanni-Vincentelli, A., Törngren, M.: Characteristics, capabilities, potential applications of cyber-physical systems: a preliminary analysis. Project Deliverable D, 2 (2013)
16. Cheng, B.H.C., Eder, K.I., Gogolla, M., Grunske, L., Litoiu, M., Müller, H.A., Pelliccione, P., Perini, A., Qureshi, N.A., Rumpe, B., Schneider, D., Trollmann, F., Villegas N.M.: Using models at runtime to address assurance for self-adaptive systems. In: Models@run.time, pp. 101–136 (2011)
17. CPSoS. Cyber-Physical Systems of Systems: Research and Innovation Priorities (2015)
18. de Lemos, R., Giese, H., Müller, H.A., Shaw, M.: Software Engineering for Self-Adaptive Systems 2. Springer, Berlin (2010)
19. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. Int. J. Softw. Tools Technol. Transfer pp. 1–21 (2015)
20. Desai, A., Seshia, S.A., Qadeer, S., Broman, D., Eidson, J.C.: Approximate Synchrony: An Abstraction for Distributed Almost-Synchronous Systems, pp. 429–448. Springer (2015)
21. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation (monographs in theoretical computer science. an eatcs series). secaucus (2006)
22. Fouquet, F., Nain, G., Morin, B., Daubert, E., Barais, O., Plouzeau, N., Jézéquel, J.-M.: An eclipse modelling framework alternative to meet the models@runtime requirements. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) Model Driven Engineering Languages and Systems, pp. 87–101, Berlin, Heidelberg. Springer Berlin Heidelberg (2012)
23. Gallagher, B.: Matching structure and semantics: a survey on graph-based pattern matching. AAAI FS **6**, 45–53 (2006)
24. Gönczy, L., Majzik, I., Bozóki, S., Pataricza, A.: MDD-based design, configuration, and monitoring of resilient cyber-physical systems. Trustworthy Cyber-Physical Systems Engineering (2016)
25. Gotz, S., Gerostathopoulos, I., Krikava, F., Shahzada, A., Spalazzese, R.: Adaptive exchange of distributed partial Models@run.time for highly dynamic systems. In: Proceedings—10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, pp. 64–70 (2015)
26. Group, O.M.: DDS for eXtremely Resource Constrained Environments, March (2019). Version 1.0
27. Hartmann, T., Fouquet, F., Jimenez, M., Rouvoy, R., Le Traon, Y.: Analyzing complex data in motion at scale with temporal graphs. In: The 29th International Conference on Software Engineering & Knowledge Engineering (SEKE'17), p. 6. KSI Research (2017)
28. Hartmann, T., Moawad, A., Fouquet, F., Nain, G., Klein, J., Traon, Y.L.: Stream my models: reactive peer-to-peer distributed models@run.time. In: 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, pp. 80–89 (2015)
29. Havelund, K.: Rule-based runtime verification revisited. Int. J. Softw. Tools Technol. Transfer **17**(2), 143–170 (2015)
30. He, X., Zhang, T., Pan, M., Ma, Z., Hu, C.-J.: Template-based model generation. Softw. Syst. Model. 1–42 (2017)
31. Hewitt, C., Bishop, P.B., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: International Joint Conference on Artificial Intelligence, pp. 235–245 (1973)
32. Horváth, Á., Varró, G., Varró, D.: Generic search plans for matching advanced graph patterns. Electronic Communications of the EASST, 6 (2007)
33. Iqbal, M.Z., Ali, S., Yue, T., Briand, L.: Applying UML/MARTE on industrial projects: challenges, experiences, and guidelines. Softw. Syst. Model. **14**(4), 1367–1385 (2015)
34. Joshi, Y., Tchamgoue, G.M., Fischmeister, S.: Runtime verification of LTL on lossy traces. In: Proceedings of the Symposium on Applied Computing - SAC '17, pp. 1379–1386. ACM Press (2017)
35. Kang, W., Kapitanova, K., Son, S.: Rdds: a real-time data distribution service for cyber-physical systems. IEEE Trans. Ind. Informat. **8**(2), 393–405 (2012)
36. Kopetz, H., Grunsteidl, G.: TTP—a time-triggered protocol for fault-tolerant real-time systems. In: *FTCS-23*, pp. 524–533 (1993)
37. Krause, C., Tichy, M., Giese, H.: Implementing graph transformations in the bulk synchronous parallel model. In: Fundamental Approaches to Software Engineering—17th xxInternational Conference, FASE 2014, Held as Part of the European Joint Confer-

ences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014, Proceedings, pp. 325–339 (2014)
38. Krupitzer, C., Roth, F.M., VanSyckel, S., Schiele, G., Becker, C.: A survey on engineering approaches for self-adaptive systems. Perv. Mob. Comput. **17**, 184–206 (2015)
39. Lee, E.A., Hartmann, B., Kubiatowicz, J., Rosing, T.S., Wawrzynek, J., Wessel, D., Rabaey, J.M., Pister, K., Sangiovanni-Vincentelli, A.L., Seshia, S.A., Blaauw, D., Dutta, P., Fu, K., Guestrin, C., Taskar, B., Jafari, R., Jones, D.L., Kumar, V., Mangharam, R., Pappas, G.J., Murray, R.M., Rowe, A.: The swarm at the edge of the cloud. IEEE Des. Test **31**(3), 8–20 (2014)
40. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebra Program. **78**(5), 293–303 (2009)
41. Maróti, M., Kusy, B., Simon, G., Lédeczi, Á.: The flooding time synchronization protocol. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, pp. 39–49. ACM (2004)
42. Mazak, A., Wimmer, M.: Towards liquid models: an evolutionary modeling approach. In: Proceedings—CBI 2016: 18th IEEE Conference on Business Informatics, Vol. 1, pp. 104–112 (2016)
43. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. Int. J. Softw. Tools Technol. Transfer **14**(3), 249–289 (2012)
44. Mitsch, S., Platzer, A.: ModelPlex: Verified runtime validation of verified cyber-physical system models. In: Intl. Conference on Runtime Verification (2014)
45. Mitschke, R., Erdweg, S., Köhler, M., Mezini, M., Salvaneschi, G.: i3QL: language-integrated live data views. ACM SIGPLAN Notices **49**(10), 417–432 (2014)
46. Morin, B., Daubert, E., Barais, O., Morin, B., Daubert, E., Barais, O.: Kevoree Modeling Framework (KMF): Efficient modeling techniques for runtime use. Technical report, University of Luxembourg (2014)
47. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: 2015 IEEE International Parallel and Distributed Processing Symposium, pp. 494–503, May (2015)
48. Nenzi, L., Bortolussi, L., Ciancia, V., Loreti, M., Massink, M.: Qualitative and quantitative monitoring of spatio-temporal properties. In: Bartocci, E., Majumdar, R. (eds.) Runtime Verification. pp, pp. 21–37. Springer International Publishing, Cham (2015)
49. Nielsen, C.B., Larsen, P.G., Fitzgerald, J.S., Woodcock, J., Peleska, J.: Systems of systems engineering: basic concepts, model-based techniques, and research directions. ACM Comput. Surv. **48**(2), 18 (2015)
50. Pardo-Castellote, G.: OMG data-distribution service: architectural overview. In: Proc. 23rd Int. Conf Distrib. Comput. Syst. Workshops (2003)
51. Peters, M., Brink, C., Sachweh, S., Zündorf, A.: Scaling parallel rule-based reasoning. In: ESWC, pp. 270–285 (2014)
52. Solis, R., Borkar, V., Kumar, P.: A new distributed time synchronization protocol for multihop wireless networks. In: Proceedings of the 45th IEEE Conference on Decision and Control, pp. 2734–2739. IEEE San Diego, USA (2006)
53. Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., Varró, D.: IncQuery-D: a distributed incremental model query framework in the cloud. In: MODELS, pp. 653–669 (2014)
54. Szárnyas, G., Izsó, B., Ráth, I., Varró, D.: The train benchmark: cross-technology performance evaluation of continuous model queries. Softw. Syst. Model. 1–29 (2017)
55. Sztipanovits, J., Bapty, T., Neema, S., Howard, L., Jackson, E.: OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems, pp. 235–248. Springer, Berlin Heidelberg (2014)
56. Sztipanovits, J., Koutsoukos, X., Karsai, G., Kottenstette, N., Antsaklis, P., Gupta, V., Goodwine, B., Baras, J.: Toward a science of cyber-physical system integration. Proc. IEEE **100**(1), 29–44 (2012)
57. Szvetits, M., Zdun, U.: Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. Softw. Syst. Model. **15**(1) (2013)
58. Teich, J., Sriram, S., Thiele, L., Martin, M.: Performance analysis of mixed asynchronous synchronous systems. In: Proceedings of 1994 IEEE Workshop on VLSI Signal Processing, pp. 103–112. IEEE (1994)
59. The Linux Foundation. Janusgraph (2018)
60. Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: an integrated development environment for live model queries. Sci. Comput. Program. **98**, 80–99 (2015)
61. Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z.: Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. Softw. Syst. Model. (2016)
62. Varró, D., Semeráth, O., Szárnyas, G., Horváth, Á.: Towards the Automated Generation of Consistent, Diverse, Scalable and Realistic Graph Models. Number 10800 (2018)
63. Varró, G., Deckwerth, F., Wieber, M., Schürr, A.: An algorithm for generating model-sensitive search plans for pattern matching on EMF models. Softw. Syst. Model. 597–621 (2015)
64. Vogel, T., Giese, H.: Model-driven engineering of self-adaptive software with EUREMA. ACM Trans. Auton. Adapt. Syst. (TAAS) **8**(4), 18 (2014)
65. Vörös, A., Búr, M., Ráth, I., Horváth, Á., Micskei, Z., Balogh, L., Hegyi, B., Horváth, B., Mázló, Z., Varró, D.: MoDeS3: model-based demonstrator for smart and safe cyber-physical systems. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 10811 LNCS, pp. 460–467 (2018)
66. Warren, D.S.: Memoing for logic programs. Commun. ACM **35**(3), 93–111 (1992)
67. Yakindu Statechart Tools. Yakindu. http://statecharts.org/
68. Zhang, B., Mor, N., Kolb, J., Chan, D.S., Lutz, K., Allman, E., Wawrzynek, J., Lee, E.A., Kubiatowicz, J.: The cloud is not enough: saving IoT from the cloud. In: 7th USENIX Workshop on Hot Topics in Cloud Computing (2015)
69. Zheng, X., Julien, C., Podorozhny, R., Cassez, F., Rakotoarivelo, T.: Efficient and scalable runtime monitoring for cyberphysical system. IEEE Syst. J. 1–12 (2016)