International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Distributed Models@run.time for Runtime Verification of Cyber-Physical Systems

Márton Búr · Gábor Szilágyi · András Vörös · Dániel Varró

Received: date / Accepted: date

Abstract Runtime models capture the operational state and context of smart cyber-physical systems (CPS) as directed, typed and attributed graphs and keep this information up-to-date in a continuously evolving environment to support various scenarios such as monitoring, control, or optimization. In Internet-of-Things applications, a CPS is frequently operating over a heterogeneous and distributed computation platform composed of resource constrained devices. The adaptation of runtime models to such scenarios has several major challenges, since (1) the runtime model needs to be distributed over multiple participants, (2) each participant may have memory limitations, and (3) the device may communicate over popular middleware platforms like the Data Distribution Service (DDS), which is a standard distributed, data-centric, publish-subscribe framework offering various quality of service guarantees.

In this paper, we first propose a time-triggered high-level model management protocol on top of the DDS middleware architecture which ensures consistent updates for distributed runtime models while offering an model manipulation interface similar to the Eclipse Modeling Framework. Moreover, we leverage DDS to manage and execute the distributed runtime models and provide fast data processing and management by elaborating a generic mapping from metamodels (captured in the Eclipse Modeling Framework) to the OMG Interface Definition Language (IDL) which is the input language for DDS. We illustrate the proposed technique using the MoDeS3 CPS demonstrator and provide an experimental evaluation.

Keywords Runtime models · Distributed model management · Data Distribution Service (DDS)

Márton Búr · Dániel Varró
Department of Electrical and Computer Engineering
McGill University, Montreal, Canada,
E-mail: marton.bur@mail.mcgill.ca, daniel.varro@mcgill.ca

Gábor Szilágyi · András Vörös · Dániel Varró
Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary

András Vörös · Dániel Varró
MTA-BME Lendület Cyber-Physical Systems Research Group, Budapest, Hungary

1 Introduction

Motivation A smart and safe cyber-physical system (CPS) [6,17,19,25,28] frequently depends on self-adaptive or autonomous behavior. Intelligent components are deployed over a heterogeneous computation platform and they continuously interact with a complex environment. Such a complexity frequently makes design time verification infeasible in practice, thus CPSs need to rely on *runtime verification* (RV) [20,23] techniques to ensure safe operation by monitoring. Recent RV approaches like [3,13] started to exploit rule-based techniques over a richer (relational or graph-based) information model.

Runtime models (aka models@ run.time [2,29]) provide such a rich knowledge representation to capture the runtime state of the domain, services and platforms in the form of typed and attributed graphs [8] to serve as a unifying semantic basis for various analysis techniques. For example, runtime models have been used for the assurance of self-adaptive systems (SAS) in [5,34] or to drive runtime verification [3] in CPS.

Problem statement In a distributed CPS, the underlying runtime model also needs to be distributed, and updated with high frequency driven by the incoming sen-

sor information and the changes in network topology. A distributed runtime model was proposed in [12] by exploiting the Eclipse Modeling Framework (EMF)[1] for specifying the models and reactive programming with lazy loading to make the complete virtual model accessible from every node over a Java-based platform. In a follow-up work [11], graphs are also versioned on a node-level granularity, allowing the users to analyze changes made to the graph over time.

However, in case of resource-constrained computational platform or strict quality of service (QoS) requirements, few guarantees are provided by existing approaches for distributed runtime models. Unfortunately, such resource constraints appear frequently in edge computing or critical embedded systems.

Objectives In [3], we presented the architecture of a framework for distributed runtime monitoring using distributed graph queries for resource-constrained CPSs, but the focus of our initial work was limited to assessing distributed query evaluation over a steady runtime model. However, in a real CPS system, the underyling runtime model is never steady, but amenable to continuous and frequent model updates. Therefore, the main objective of our current work is to provide support for such rapidly changing distributed runtime models.

Our *distributed runtime model* is built on top of an EMF-compatible model manipulation interface and a semantically well-founded, time-triggered high-level model update protocol that runs over a distributed and standard middleware platform that manages resource constrained devices as participants.

To provide QoS guarantees for communication between components of the platform, we incorporate the Data Distribution Service standard [26] as a reliable underlying messaging middleware. As such, reliable message delivery is handled at a lower abstraction layer.

In principle, the publish-subscribe approach used in DDS is independent of the underlying data model, but in practice, integrating DDS with a particular application requires significant manual programming due to the low-level APIs and the different programming style imposed by the publish-subscribe principle. For this reason, we provide a mapping to the DDS architecture to automatically synthesize the configuration settings for the communication infrastructure. As such, we combine mature technologies from modeling languages and distributed systems into a novel, high-level model management and query framework which can be used for distributed runtime verification [3].

We illustrate our concepts in the context of the MoDeS3 CPS demonstrator [35], and provide a prototype implementation of the framework. Furthermore, we carried out an initial scalability evaluation our prototype in the context of the MoDeS3 demonstrator.

This paper extends our initial work [3] by providing

(i) a semantically well-founded model update protocol
(ii) a mapping of metamodel concepts and query definitions to the standard DDS architecture,
(iii) a prototype implementation,
(iv) a novel performance evaluation which covers the runtime model update phase.

2 Overview of Query-based Distributed Runtime Monitoring

Figure 1 is an overview of distributed runtime monitoring of CPSs deployed over a heterogeneous computing platform with multiple participants using distributed runtime models and graph queries. The current paper will elaborate on the distributed runtime model.

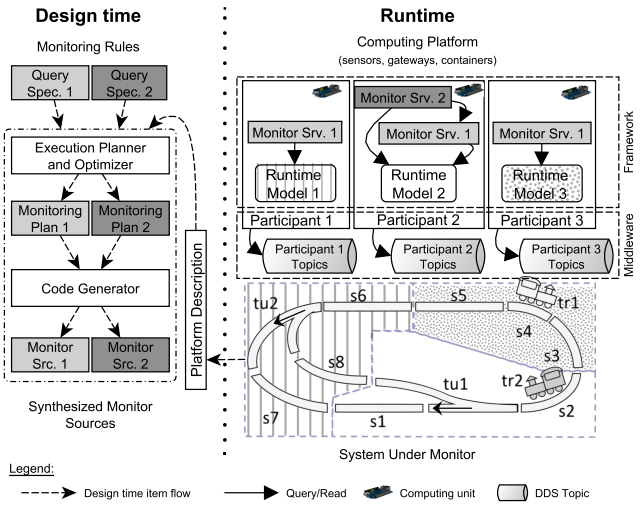

Fig. 1: Distributed runtime monitoring by graph queries

During the system 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. The 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. Then this execution plan is passed on to the code generator to produce platform dependent C++ source code, which is ready to be compiled into an executable for the plat-

[1] `http://www.eclipse.org/emf`

form. To provide better focus for the current paper, this component will not be detailed here.

Runtime monitor programs are *deployed to a distributed heterogeneous computation platform*, which is assumed to include various types of computing units (referred to as *participants*) ranging from ultra-low-power microcontroller units, through smart devices to 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 fast and reliable communication between the components.

The model management 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 [15]). RDDS is an extension for the DDS standard [26] 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 [2, 29], 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.

Running example. We illustrate distributed runtime models in the context of the MoDeS3 demonstrator (*Model-Based Demonstrator for Smart and Safe Cyber-Physical Systems*) [35], 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 Figure 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. A participant may read its local sensors, (e.g., the occupancy of a segment, or the status of a turnout), 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

We revisit definitions related to metamodeling, and runtime models. The section also provides a brief overview on the Data Distribution Service standard [26].

3.1 Domain-specific modeling languages

Many industrial CPS modeling tools build on the concepts of domain-specific (modeling) languages (DSLs) 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, 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 1 Figure 2 shows a metamodel for the MoDeS3 demonstrator with Participants (identified in the network by hostID attribute) which host the DomainElements. A DomainElement is either a Train or RailroadElement. A Train has a speed attribute and the train is always located on a RailroadElement. Turnouts and Segments are RailroadElements with links to the left and right side RailroadElements. These left and right references are used to describe the actual connections between the different RailroadElements. These references

are navigable in both directions. A Turnout has references to RailroadElements that represent the straight and divergent directions.

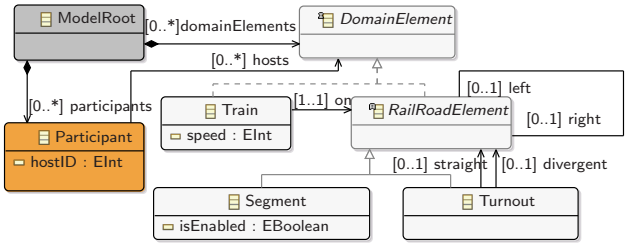

Fig. 2: Metamodel for MoDeS3

Structural consistency constraints References in EMF metamodels may contain multiplicity constraints *lo..up* which consists 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 modles), while 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 Figure 2). EMF maintains such pairs of opposite references consistently in case of non-distributed instance models, i.e. if such 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.

3.2 Runtime models

The objects, their attributes, links, and dependencies between objects constitute a runtime knowledge base for the underlying system in operation called a *runtime model* [2, 29]. Relevant changes in the system are reflected in the runtime model (in an event-driven or time-triggered way) and operations executed on the runtime model (e.g. setting values of controllable attributes or relations between objects) are reflected in the system itself (e.g. by executing scripts or calling services). We assume that this runtime model is self-descriptive in the sense that it contains information about the computation platform and the allocation of services to platform elements, which is a key enabler for self-adaptive systems [5, 34].

A *runtime model* $M = \langle Dom_M, \mathcal{I}_M \rangle$ is a 2-valued logic structure over Σ, as in [32], 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 a 2-valued 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.
- **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.
- **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.

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* 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 is stored by host p.

Example 2 Figure 3 shows a snapshot of the distributed runtime model M_d for the MoDeS3 system depicted in the right part of Figure 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 Figure 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.

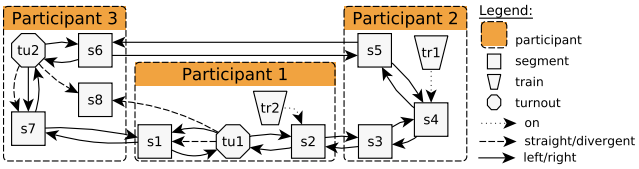

Fig. 3: Distributed runtime model for MoDeS3

3.4 Model update operations

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

- **Object operations**: In regular (regular) runtime models, objects can be *created* and *deleted*. In a distributed setting, objects can also be *moved* from one participant to another. Object update operations are always with 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 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.

We do not support *reference move* as an atomic (single) update operation. Instead, an explicit deletion and an explicit creation operations are needed. This implies that for references with multiplicity 0..1, adding a link will be rejected until the current link is present.

3.5 Real-time Data Distribution Service

The OMG specification for DDS [26] provides a common application-level interface for data-centric implementations over a *publish-subscribe* communication model. Additionally, this specification defines the main features suitable for applying in embedded self-adaptive systems. We provide a brief DDS overview based on [26].

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

A simplified metamodel of DDS implementing the publish-subscribe model is depicted in Figure 4. Participant is the top-level entity in a DDS application, so we assume that each deployed program creates exactly one instance, 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 there is a new state for a data object available, i.e., invoking write_w_timestamp() will not necessarily cause immediate communication. Similarly, the task of a Subscriber is to decide when to invoke the DataReader.take() method that does the actual reading of new data values.

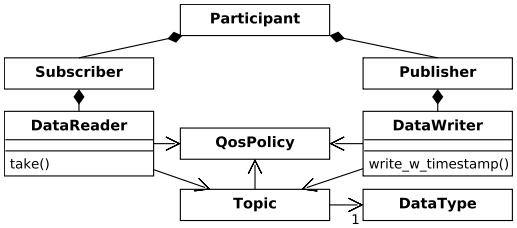

Fig. 4: UML class diagram of DDS classes

Unlike traditional publish-subscribe protocols[2][3], 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 of data objects in the global data space. Additionally, this coupling between Topic and DataType (with the additional QosPolicy settings) enables implementation optimizations such as pre-allocating the resources needed to send or receive messages of a Topic.

Real-time DDS by [15] is an extension of the DDS standard, which tailors DDS to fit 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

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

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.

4 A Model Management Protocol for Distributed Runtime Models

Next, we extend our previous work [3] to provide a real-time and distributed runtime model management protocol built over a reliable communication middleware.

4.1 Overview

Our work addresses decentralized mixed synchronous systems where participants (1) communicate model updates to other participants at the beginning of a time-triggered execution loop [16] (*update cycle*) and(2) then evaluate monitoring queries over a consistent snapshot of the system (*query cycle*). We focused on the query cycle in [3] while this paper focuses on the model update cycle. We assume approximate synchrony [7] between the clocks of individual computing units, thus all messages should arrive within given t timeframes reserved for the update or query phases.

We assume that each participant may locally receive model update request (e.g. from a sensing service) either asynchronously or periodically at any point in time, but such a model update request is registered (with a timestamp) but buffered to be processed later in a time-triggered way by the distributed runtime model. The real processing order of model updates will not be time-ordered, but our protocol provides global consistency guarantees (see later in subsection 4.7) periodically at the end of model update cycle. As such, distributed graph queries used for runtime monitoring [3] will execute over a consistent snapshot of the system.

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 DDS (i.e. no messages are lost or get delayed).
2. If there is still a violation of QoS guarantees, the middleware notifies participants to allow dynamic reconfiguration (i.e. fault-tolerance is provided on the communication-level, but not on the model-level)
3. The synchrony of local physical clocks of computing units is enforced by a clock synchronization protocol [21,27], thus each participant receives messages with a timestamp marking the time of the update action was initiated.
4. Participants can save such update messages to a pre-allocated cache with potentially limited size. This way, participants operating 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.

Potential race conditions. Despite we assume that there is a single source of truth for every model object, there are still several potential race conditions which necessitates to exchange messages between participants in order to maintain a structurally consistent model (at the end of the update cycle).

- One participant may wish to move an object and the previous host of the object would like to execute any local update operation on that object (e.g. delete the object, set an attribute or update an outgoing reference) within the same cycle.
- A more subtle case is when two participants wish to add a reference to the same target object, but this reference also has an inverse reference with at most one multiplicity. In such a case, only one of the references should allowed to be set otherwise the model becomes inconsistent.

4.2 A multi-phase model update protocol

In our distributed model update protocol, *object creation, move and deletion are communicated as broadcast messages* so that participants can register all objects. Such broadcast messages allow each participant to add or remove references to any model object even if the model objects are not hosted by the participant. On the other hand, *messages for reference addition and removal are sent in a peer to peer manner.*

The protocol consists of three major conceptual phases which are discussed in the following: (1) object update, (2) reference update, (3) reference acknowledgement.

4.2.1 Object update phase

Object creation phase The first phase of the model update cycle addresses object creation. A participant must send a *broadcast message* with the identifier o_{create}, the type `C`, and the host p_{host} of the added object. Formally, the sent message has $[\![\texttt{C}(o_{create})]\!]^{M_d}@p_{host} = 1$

as content. It is necessary to notify other participants about the addition of a new model element in order to allow them to create references pointing to the object (i.e. as a target end of an edge). Recipient participants will create a *proxy object* locally.

Object move phase A special case of object updates is moving objects between hosts. Formally, this is an object creation message that initiates to move an existing object to a new host participant. Given an object with identifier o_{move}, this is achieved by sending a message with the content $[\![\texttt{C}(o_{move})]\!]^{M_d}@p_{new\_host} = 1$. It is worth mentioning that upon moving an object, only proxy objects have to be updated, but incoming references pointing to the object do not need updates, since the identifier of the moved object does not change. Table 1 summarizes the actions to be taken by participants upon receiving a *create object* message.

Object delete phase. In the second phase of object updates, objects of the runtime model can be deleted. The phase is similarly to object creation: the identifier o_{delete}, the type C, and the host p_{host} of the deleted object is sent in a *broadcast message*. Formally, the sent message has $[\![\texttt{C}(o_{delete})]\!]^{M_d}@p_{host} = 0$ as content, 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 removed 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 non-reversible, i.e., once an object is deleted, it cannot be recreated. Table 2 summarizes the actions to be taken upon receiving a *delete object* message.

4.3 Reference update phase

In this is second phase of the model update protocol, link additions and removals are initiated (in arbitrary order) between objects in this phase.

Link addition The addition of a link from object o_{src} to o_{trg} is carried out 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 the structural consistency checks can be done locally by the host of o_{src}.

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 content $[\![\texttt{R}(o_{src}, o_{trg})]\!]^{M_d}@p_{src} = 1$. In order to maintain a consistent model, the local knowledge base keeps $[\![\texttt{R}(o_{src}, o_{trg})]\!]^{M_d}\ @p_{src} = 0$ entry until receiving an acknowledging reply from the host of the target object containing $[\![\texttt{R}(o_{src}, o_{trg})]\!]^{M_d}@p_{trg} = 1$. However, if the reference cannot be added for some reason, such as a multiplicity constraint would be violated, the reply from p_{trg} will be $[\![\texttt{R}(o_{src}, o_{trg})]\!]^{M_d}@p_{trg} = 0$. From this information the host of o_{src} will also deduct that the reference cannot be set, thus a consistent truth value is maintained by both parties.

Link deletion 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, deleting 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 deleting a reference of type R, the content of the messages is $[\![\texttt{R}(o_{src}, o_{trg})]\!]^{M_d}@p_{src} = 0$. Meanwhile, in the local knowledge base, this entry is stored until acknowledgement arrives.

4.4 Reference acknowledgement phase

Special attention is needed to handle the update of inverse references (which need to be added and removed simultaneously) with 0..1 multiplicities due to the potential race condition between participants. Thus, when a reference with an opposite is changed, the host of the target object needs to acknowledge the operation for the host of the source object in a subsequent *reference acknowledgement phase*.

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

Table 3 and Table 4 briefly summarize the actions to be taken by participants when they receive reference update messages for inverse references.

4.5 Cycled processing by transaction logs

As updates to the system may occur while in operation, each participant maintains a *transaction log* (buffer) to

Table 1: Summary of actions for message $[\![\mathtt{C}(obj)]\!]^{M_d}@p = 1$ (creating/moving an object)

	Participant	
Condition	p (sender)	other (receiver)
Model object *obj* is unknown	create local *obj*	create proxy for *obj*
A proxy for *obj* is present locally	replace proxy with local object *obj*	update proxy for *obj*
The object *obj* is present locally	no-op	replace local object *obj* with a proxy

Table 2: Summary of actions for message $[\![\mathtt{C}(obj)]\!]^{M_d}@p = 0$ (deleting an object)

	Participant	
Condition	p (sender)	other (receiver)
Model object *obj* is unknown	no-op	no-op
A proxy for *obj* is present locally	remove proxy and any dangling links	remove proxy and any dangling links
The object *obj* is present locally	delete *obj* and any dangling links	delete *obj* and any dangling links

Table 3: Summary of actions for message $[\![\mathtt{R}(src, trg)]\!]^{M_d}@p = 1$ (adding a link with opposite)

	Participant	
Condition	p_{src}, host of *src* (sender)	p_{trg}, host of *trg* (receiver)
link already exists	no-op	send $[\![\mathtt{R}(src, trg)]\!]^{M_d}@p_{trg} = 1$ as reply to p_{src}
link does not exist	await reply and add link if acknowledged	send $[\![\mathtt{R}(src, trg)]\!]^{M_d}@p_{trg} = 1$ or 0 as reply to p_{src} and add opposite link if 1 is sent back

Table 4: Summary of actions for message $[\![\mathtt{R}(src, trg)]\!]^{M_d}@p = 0$ (removing a link with opposite)

	Participant	
Condition	p_{src}, host of *src* (sender)	p_{trg}, host of *trg* (receiver)
link exists	remove link	remove opposite link
link does not exist	no-op	no-op

register such changes. Within the same update cycles, participants need to filter out cancelled or overwritten model update actions prior to communicating them to other participants (e.g. if the reference is update twice). This pre-filtering of update actions is an important step that guarantees that a given predicate evaluation is only communicated once in a model update phase.

Then, at the beginning of the model update phase, only the (filtered) changes that arrived during the time window of the previous cycle will be considered and communicated by each participant to the distributed runtime model. This way, a globally consistent snapshot of the system is obtained by the end of the model update cycle wrt. the changes arrived until the beginning of the same model update cycle. Changes arriving during the current model update (and query evaluation) cycle will be processed in the next cycle.

Example 3 Figure 5 shows a possible timeline for two participants with their corresponding transaction logs with all update actions marked for processing. When processing this queue (right part), each participant sends messages following the order described in this section previously (object create, object delete, reference update, reference add, and reference remove) for each update that does not only effect the local knowledge base. For instance, the update with timestamp $t_{1,2}$ is marked as local and will not incur communication, since both ends of the reference are hosted by p_1. Moreover, participants send messages of the same kind respecting the ordering between the timestamps of the corresponding update actions.

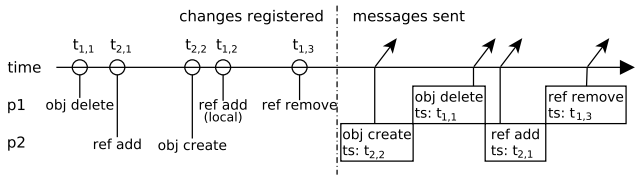

Fig. 5: Registering changes and communicating updates

4.6 Handling of object move

The fact that messages conveying the information on updates are in a different order as the update actions in the transactions logs can cause *delayed updates*. This, in most cases, requires no extra consideration, since this reordering is carried out it help prevent ambiguous cases during model update. However, a race condition may arise when an object receives a delayed update that has been moved between two participants within the same model update cycle.

Figure 6 depicts a case when a participant p_1 registers an *update* to a hosted object obj that is moved shortly after it to be hosted by p_2. In this case, the timestamps wrt. a globally synchronized clock are $t_{1,i} < t_{2,j}$. When participants communicate changes, first p_2 sends a broadcast message about the moving of the object. Later p_1 sends an update regarding obj. The message on the update is surely sent later by p_1 than the message about the move, since object create and move messages are communicated first and p_1 is not sending an object create. In this case, any participant except for p_2, that receive the update message sent by p_1 will ignore the message, since the object obj p_1 is trying to update is no longer hosted by p_1. However, when p_2 receives the message, it will execute the update to the object and resend the message with the same content because the update to the object happened earlier (in the global timeline) that the move operation.

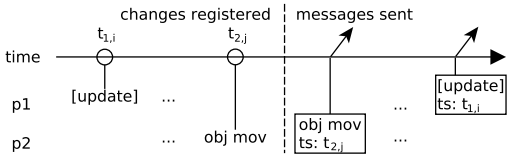

Fig. 6: Accepted delayed update ($t_{1,i} < t_{2,j}$)

Moreover, Figure 7 shows a case when the opposite is happening. Again, the same messages are sent in the same order as before, but this time the global timestamps are $t_{1,i} > t_{2,j}$. This way, the recipient p_2 will know that the update happened after the actual moving of obj, thus p_1 no longer had rights to apply updates, and p_2 refuses to change the state of the object.

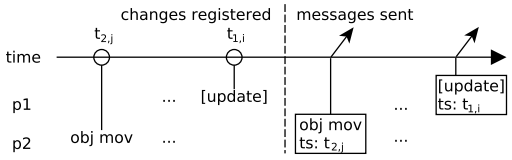

Fig. 7: Refused delayed update ($t_{1,i} > t_{2,j}$)

4.7 Semantic guarantees

Termination. To show termination of the protocol, we claim that model update message sequences are free from deadlocks and livelocks. Recall that (1) local updates may only touch a finite number of predicates (and they are assumed to complete in short time), and (2) the distributed model update protocol sends messages either in a "send and forget" or in a "request and acknowledge" manner.

Theorem 1 *The distributed model update protocol is free from deadlocks and livelocks.*

Proof (Sketch) To show the deadlock-free property, it is sufficient to show that reply messages will always arrive, which is ensured by the QoS guarantees of the underlying DDS middleware, and by our assumption that all local updates to the knowledge base touch only finite elements.

Livelock-freedom can be proved by showing that (i) messages that do not require replies do not trigger recipients to send further messages and (ii) messages sent in a request-acknowledge manner require a single reply message while reply messages are not generating additional messages. As a result, communicating an update operation involves sending at most two messages.

Consistency. We claim that the provided distributed model update protocol is consistent, i.e., by the end of the model update cycle, the set of predicates gained by fusing each participant's knowledge base is free from contradictions, and provides the same effect as if the model updates were processed by a centralized global repository.

Let $\{ac^i(t_{i,1}), ac^i(t_{i,2}), \ldots, ac^i(t_{i,k})\}$ be transaction logs for each participant i and a given time window $(0, t_{upd})$ where (1) $t_{i,j}$ is a globally unique timestamp for the corresponding model update action (i.e., $t_{i,j} = t_{x,y} \rightarrow i = x \wedge j = y$) and (2) each participant's transaction log is locally time-ordered: $0 < t_{i,j} < t_{i,j+1} < t_{upd}$.

Let $Seq_{p_i,loc} : M_0 \rightsquigarrow M_l$ be the time-ordered sequence of model update actions logged by participant p_i between two consecutive model update phases (applied on initial model M_0 and yielding model M_l as result) and let $Seq_{p_i,msg} : M_0 \rightsquigarrow M_m$ be the sequence of executed model update actions by p_i in the distributed runtime model during the model update phase (applied on initial model M_0 and yielding model M_m as result).

Theorem 2 (Local consistency of model updates) *For any atomic predicate ψ the effect of model updates made by a single participant is time-serializable, i.e., for any $Seq_{p_i,loc} : M_0 \rightsquigarrow M_l$ and the corresponding message sequence $Seq_{p_i,msg} : M_0 \rightsquigarrow M_m$ derived by the protocol: $[\![\psi]\!]^{M_l} = [\![\psi]\!]^{M_m}$.*

Let $Seq_{glo} : M_0 \rightsquigarrow M_g$ be the globally time-ordered sequence of model update actions logged by a hypothetical centralized participant between two consecutive model update phases (applied on initial model M_0 and yielding model M_g as result) and let $Seq_{drm} : M_0 \rightsquigarrow M_d$ be the sequence of executed model update actions in the distributed runtime model during the model update phase (applied on initial model M_0 and yielding model M_d as result).

Theorem 3 (Global consistency of model updates) *For any atomic predicate ψ the effect of model updates is globally time-serializable, i.e., for any $Seq_{glo} : M_0 \rightsquigarrow M_g$ and its corresponding $Seq_{drm} : M_0 \rightsquigarrow M_d$ derived by the protocol: $[\![\psi]\!]^{M_d} = [\![\psi]\!]^{M_g}$.*

5 Synthesis of Middleware Configurations

Next, we introduce a mapping from high-level runtime models to the input language of the DDS runtime, so that we can automatically synthesize DDS configurations for applications. We are focusing on *data types*, *topics*, *publishers*, *subscribers*, and *participants* for now, but this could be extended to other parameters as well, such as refined QoS settings. We use IDL [18] configuration language to capture the necessary definitions.

5.1 From metamodel elements to IDL types

First, domain classes of the metamodel are mapped to literals of enumeration `ObjectType`. Similarly, the enumeration `ReferenceType` is created with a literal for each reference in the metamodel following the pattern *≪SourceClassName≫_≪ReferenceName≫*.

Example 4 Listing 5.1 depicts two enumerations `ObjectType` (lines 1-9) and `ReferenceType` (lines 10-19) generated from the metamodel of Figure 2.

```
1  enum ObjectType {
2    ModelRoot,
3    Participant,
4    DomainElement,
5    RailRoadElement,
6    Train,
7    Segment,
8    Turnout
9  };
10 enum ReferenceType {
11   ModelRoot_domainElements,
12   ModelRoot_participants,
13   Participant_hosts,
14   RailRoadElement_left,
15   RailRoadElement_rigt,
16   Train_on,
17   Turnout_straight,
18   Turnout_divergent
19 };
```

Listing 5.1: Generated IDL type definitions for the example metamodel

Then, we provide the definitions of generic message data types for DDS communication, as shown in Listing 5.2 with corresponding structures `ObjectUpdate` (lines 5−10), `ReferenceUpdateRequest` (lines 11−17), and `ReferenceUpdateReply` (lines 18 − 24).

```
1  enum UpdateType {
2    Create,
3    Delete
4  };
5  struct ObjectUpdate {
6    long long id;
7    ObjectType oType;
8    UpdateType uType;
9    long long senderPID;
10 };
11 struct ReferenceUpdateRequest {
12     long long srcID;
13     long long trgID;
14     ReferenceType rType;
15     UpdateType uType;
16     long long senderPID;
17 };
18 struct ReferenceUpdateReply {
19     long long srcID;
20     long long trgID;
21     ReferenceType rType;
22     boolean isSuccessful;
23     long long senderPID;
24 };
```

Listing 5.2: IDL definition of generic model update types

An `ObjectUpdate` message uniformly handles the case for object creation and deletion. Such a message includes the unique ID of the object it is referring to (`id`), its type (`oType`), the update operation type (`uType`), and the sender participant ID (`senderPID`). The `id` is a `long` attribute that is assumed to be globally unique in the knowledge base. The value assigned to `oType` is

one of the generated values coming from the domain metamodel to add metadata about the type of the object being changed. The possible values of `uType` capture the different update operations that are defined in lines $1-4$. The currently supported update operations are delete or create/move, but other update types could be incorporated similarly. Finally, `senderPID` identifies the participant who updates the object.

A `ReferenceUpdateRequest` message is similar to `ObjectUpdate` but with two identifiers, `srcID` and `trgID`, denoting the source and reference target object of the reference. The attribute `rType` may have values from an enumeration with literals generated from the associations found in the metamodel. To acknowledge when a new reference is added, a `ReferenceUpdateReply` type is generated, where the `isSuccessful` Boolean flag indicates whether the addition was successful or not.

5.2 Participants and topics

In addition to deriving data types from the domain model, we show how applications can use them to set up the DDS middleware. Currently, each application manages a single participant connected to the same (default) domain. Additionally, we use a single publisher and a subscriber that is automatically provided for each participant. DDS data writers and data readers will be attached to this publisher and subscriber, respectively.

A general broadcast topic (`/Object`) delivers object deletion and creation messages. Each participant is subscribed to this topic as well as writes this topic. Broadcast messages are of type `ObjectUpdate` which is thus the assigned data type of this topic.

To communicate bidirectional referenece updates, each participant subscribes to two topics that other participants can use to send (direct) unicast messages to them. These topics are `/ParRefRequest_n` and `/ParRefReply_n`, where `n` represents the ID of the participant. The DDS data types assigned to these topics are `ReferenceUpdateRequest` and `ReferenceUpdateReply`.

These message types and middleware setup enable to execute the distributed model update protocol detailed in subsection 4.2 The number of DDS entities (participants, topics, data writers and data readers) in the system is independent from the size of the model and also from the size of the metamodel, which is a desirable property in DDS [26]. The total number of topics is $2 * \#\{\text{participants}\} + 1$. All participants are uniquely addressable and they will have their own topics: the DDS protocol manages the communication to be real-time and reliable between the participants.

6 Evaluation

We conducted measurements to evaluate and address the research questions:

> ***Q1**: How does our approach scale with respect to model size?*
> ***Q2**: How does our approach scale with respect to the number of participants?*

6.1 Measurement setup

Computation platform. We reserved 20 on-demand Ubuntu Server 18.04 LTS t2.micro instances in the Amazon AWS Cloud to run our benchmarks. Each machine was assigned 1 (virtual) CPU core and 1 GB of RAM which represents a single board embedded device. This way we are able to assess the scalability of our distributed runtime model approach wrt. the number of participants operating on the same network and to scale up models to reasonable large sizes.

DDS middleware. We used the DDS implementation created by RTI[4] which supports the quality of service settings included in the DDS specification. Furthermore, RTI provides additional options to fine-tune applications. We slightly modified an 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 RIT's own extensions to the standard.

CPS runtime model benchmark. 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 the CPS demonstrator 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 [14].

We defined a model *template* (or a *building block*) for the model generator with a total of 14 connected domain objects (five turnouts, five segments, three trains, and one participant). Each block is set up in a way that the three pre-selected Turnout model objects can have references to up three objects contained in other blocks via their left or right references.

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

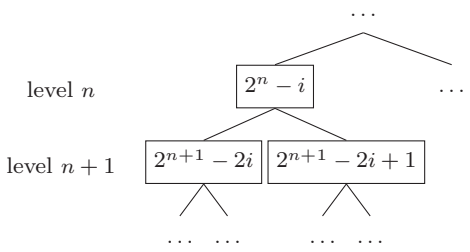

Fig. 8: Relationship between block indices during model generation

At model generation time, participants are instantiating such blocks with unique identifiers using the formula of $ID = j + k \cdot P$, where j $(0 \leq j < P)$ denotes the unique participant ID , k is the k^{th} block created by that participant, and P is the total number of participants in the system. Adding references between objects contained in different blocks follows the scheme depicted in Figure 8. A participant obtains n and i parameters using $ID = 2^n - i, i < 2^{n-1}$ and calculates identifiers for targeted blocks with identifiers $2^{n+1} - 2i$ and $2^{n+1} - 2i + 1$. As a final step, it adds references between turnouts defined by the template.

6.2 Measurement results

6.2.1 Model update throughput.

In the first set of experiments, we assessed how the model update throughput is affected by the number of participants present the system. Each participant was sending the same amount of broadcast update messages (each creating 50K new objects), while also listening to model updates sent by other participants.

Figure 9 shows our results. Each line represents a separate scenario where 2, 5, 10, and 20 participants were active, 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).

This Figure 9 suggests that the throughput of model updates is unaffected both by the actual size of the model and the number of participants. Additionally, the results also point out that our approach *scales up to 1M model objects hosted across 20 participants.*

We also assessed throughput for different model update types. Figure 10 shows the results for measuring the capability of a single participant to process various model update messages. On average, 1708 object is registered every 10 ms, while this value is only 286 and 462 for processed reference update requests and replies, respectively, which are promising performance indicators for a software prototype.

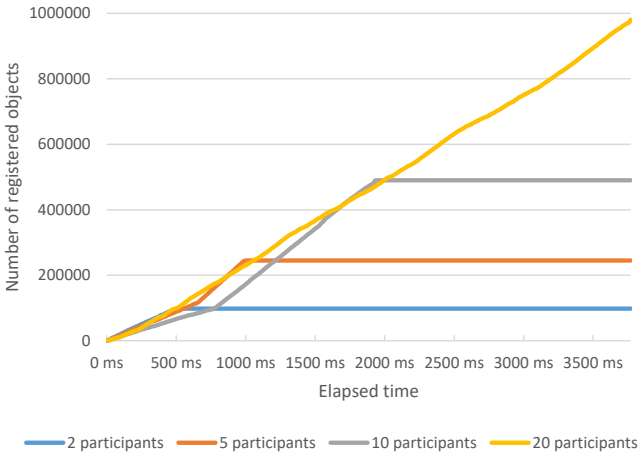

Fig. 9: Number of model objects registered by a single participant

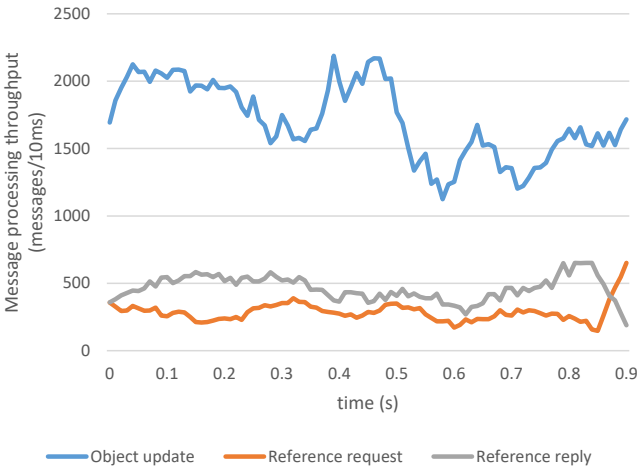

Fig. 10: Throughput comparison for processing different model updates by a single participant

6.2.2 Model update intervals.

According to our time-triggered model management approach, participants operate according to a global synchronized clock, and messages should arrive within a given timeframes for successful processing. For this reason, we conducted measurements to give initial estimations on the length of time intervals that are required to communicate a given number of changes. In this scenario, we used a fixed number of 20 participants, but each communicated a given number of equal changes periodically (overall global model deltas were of size 1K, 10K and 100K) while generating a model of given size (approximately reaching 0.5M elements). Each participant logged the event when received all messages of

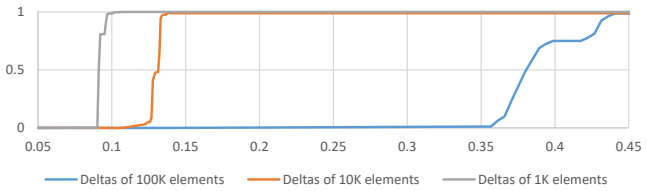

Fig. 11: Distribution of model update time duration for different change sizes

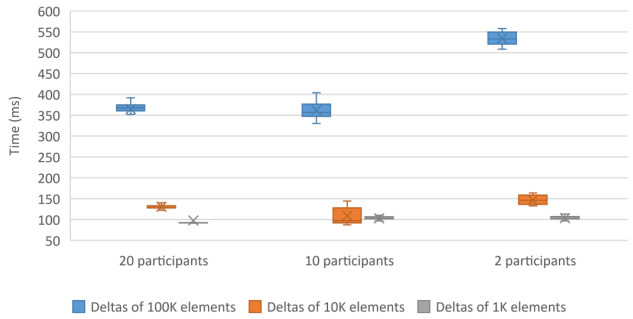

Fig. 12: Time needed to apply object deltas of different sizes

a model update delta. The empirical distribution functions obtained for the different change sizes are shown in Figure 11.

This benchmark revealed that 1K (max measured duration: 107 ms) and 10K (max: 140 ms) changes require almost the same amount of time to propagate with a very slight variance. On the contrary, larger changes affecting 100K elements (max: 450 ms) took almost 4× longer to propagate and had greater variance.

6.2.3 Communication costs.

Another set of experiments was addressing sending model deltas of given sizes to determine the impact of change sizes on the communication, while also changing the number of participants in the system. Figure 12 presents the obtained results.

A possible explanation for the results is that communicating a given number of changes requires proportionally more resources in case there are fewer participants in the system. While the total number of changes to the model is the same for each scenario in a given time window, the middleware seems to be able optimize resource utilization better.

6.2.4 Threats to validity.

The generalizability of our experimental results is limited by certain factors. First and foremost, we reserved simple instances in the Amazon AWS 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 traffic on the physical network while performing our evaluations was not known. Additionally, to measure the performance of our distributed model management approach, participants executed only model management tasks. Finally, it is important to add that while assessing model update throughput for reference updates, the change sizes for references that implied communication were potentially not equal for different participants.

7 Related Work

Runtime models. The models@ run.time paradigm [2] serves as the conceptual basis for the Kevoree framework [24]. This framework originally aimed at providing an implementation and adaptation of the de facto EMF standard for runtime models [9]. 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 depends on the Java programming language and the Eclipse modeling framework, which questions its applicability to resource-constrained environments.

The work presented in [12] 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 Plane [37]. 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 in this cases represented by time series logs, which is considered as low-level representation compared to graph models.

Another distributed solution for the models@ run.-time is presented in [34] 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 was studied in [10]. The authors propose a role-based model synchronization approach for efficient knowledge sharing. First, they identify three strategies for model synchronization wrt. the participants share what part of their knowledge base. Then, with the help of different roles, they show optimizations for knowledge sharing in terms of performance, energy consumption, memory consumption, and data privacy, while in our approach data ownership is exclusive and based on the platform.

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) [30]. 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 [4] 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 [11] introduce GreyCat, an implementation for *temporal graphs*. By adding timestamps to graph node IDs, 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 out approach, they use a per-node locking approach to prevent inconsistencies.

Finally, it is worth pointing out that 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 [1], while in [22] DDS is suggested as a key enabler technology for allowing timely and reliable data delivery for modern model-based applications.

8 Conclusions

In this paper, we proposed a time-triggered and distributed runtime model management approach built on top of the standard DDS reliable communication middleware that is widely used in used in self-adaptive and resource constrained CPS applications. The main use case for such distributed runtime models is to carry out runtime verification by exploiting graph query techniques as detailed previously in [3].

Our approach introduces an efficient handling of a distributed knowledge base stored in as a graph over a heterogeneous computing platform. Consistent manipulation and update of the knowledge base is defined as a distributed and time-triggered model management protocol and implemented with the help of 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.

Acknowledgements This paper is partially supported by MTA-BME Lendület Cyber-Physical Systems Research Group, the NSERC RGPIN-04573-16 project, the Werner Graupe International Fellowship in Engineering (as part of the MEDA program), and the ÚNKP-17-2-I New National Excellence Program of the Ministry of Human Capacities.

References

1. Luciano Baresi and Carlo Ghezzi. The disappearing boundary between development-time and run-time. In *FoSER*, 2010.
2. Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.
3. Márton Búr, Gábor Szilágyi, András Vörös, and Dániel Varró. Distributed Graph Queries for Runtime Monitoring of Cyber-Physical Systems. In *21st International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 111–128, 2018.
4. CallidusCloud. Orientdb, 2018.
5. Betty H. C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas. Using models at runtime to address assurance for self-adaptive systems. In *Models@run.time*, pages 101–136, 2011.
6. Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. *Software Engineering for Self-Adaptive Systems 2*. Springer, 2010.
7. Ankush Desai, Sanjit A. Seshia, Shaz Qadeer, David Broman, and John C. Eidson. *Approximate Synchrony: An Abstraction for Distributed Almost-Synchronous Systems*, pages 429–448. Springer, 2015.
8. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamentals of algebraic graph transformation (monographs in theoretical computer science. an eatcs series). secaucus, 2006.
9. François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. An eclipse modelling framework alternative to meet the models@runtime requirements. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Sys-*

tems, pages 87–101, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

10. Sebastian Gotz, Ilias Gerostathopoulos, Filip Krikava, Adnan Shahzada, and Romina Spalazzese. Adaptive exchange of distributed partial Models@run.time for highly dynamic systems. *Proceedings - 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015*, pages 64–70, 2015.
11. Thomas Hartmann, Francois Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. Analyzing complex data in motion at scale with temporal graphs. In *The 29th International Conference on Software Engineering & Knowledge Engineering (SEKE'17)*, page 6. KSI Research, 2017.
12. Thomas Hartmann, Assaad Moawad, François Fouquet, Grégory Nain, Jacques Klein, and Yves Le Traon. 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*, pages 80–89, 2015.
13. Klaus Havelund. Rule-based runtime verification revisited. *Int. J. Softw. Tools Technol. Transfer*, 17(2):143–170, 2015.
14. Xiao He, Tian Zhang, Minxue Pan, Zhiyi Ma, and Chang-Jun Hu. Template-based model generation. *Software & Systems Modeling*, pages 1–42, 2017.
15. Woochul Kang, Krasimira Kapitanova, and Sh Son. Rdds: A real-time data distribution service for cyber-physical systems. *IEEE Trans. Ind. Informat.*, 8(2):393–405, 2012.
16. H. Kopetz and G. Grunsteidl. TTP - a time-triggered protocol for fault-tolerant real-time systems. In *FTCS-23*, pages 524–533, 1993.
17. Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems. *Perv. Mob. Comput.*, 17:184–206, feb 2015.
18. David Alex Lamb. Idl: Sharing intermediate representations. *ACM Trans. Program. Lang. Syst.*, 9(3):297–318, July 1987.
19. Edward A Lee, Björn Hartmann, John Kubiatowicz, Tajana Simunic Rosing, John Wawrzynek, David Wessel, Jan M Rabaey, Kris Pister, Alberto L Sangiovanni-Vincentelli, Sanjit A Seshia, David Blaauw, Prabal Dutta, Kevin Fu, Carlos Guestrin, Ben Taskar, Roozbeh Jafari, Douglas L Jones, Vijay Kumar, Rahul Mangharam, George J Pappas, Richard M Murray, and Anthony Rowe. The swarm at the edge of the cloud. *IEEE Design & Test*, 31(3):8–20, 2014.
20. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
21. Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 39–49. ACM, 2004.
22. Alexandra Mazak and Manuel Wimmer. Towards liquid models: An evolutionary modeling approach. *Proceedings - CBI 2016: 18th IEEE Conference on Business Informatics*, 1:104–112, 2016.
23. Stefan Mitsch and André Platzer. ModelPlex: Verified runtime validation of verified cyber-physical system models. In *Intl. Conference on Runtime Verification*, 2014.
24. Brice Morin, Erwan Daubert, Olivier Barais, Brice Morin, Erwan Daubert, and Olivier Barais. Kevoree Modeling Framework (KMF): Efficient modeling techniques for runtime use. Technical report, University of Luxembourg, 2014.
25. Claus Ballegaard Nielsen, Peter Gorm Larsen, John S Fitzgerald, Jim Woodcock, and Jan Peleska. Systems of systems engineering: Basic concepts, model-based techniques, and research directions. *ACM Comput. Surv.*, 48(2):18, 2015.
26. Gerardo Pardo-Castellote. OMG Data-Distribution Service: Architectural overview. In *Proc. 23rd Int. Conf Distrib. Comput. Syst. Workshops*, 2003.
27. Roberto Solis, Vivek Borkar, and PR Kumar. A new distributed time synchronization protocol for multihop wireless networks. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 2734–2739. IEEE San Diego, USA, 2006.
28. J. Sztipanovits, X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, B. Goodwine, and J. Baras. Toward a science of cyber-physical system integration. *Proceedings of the IEEE*, 100(1):29–44, 2012.
29. Michael Szvetits and Uwe Zdun. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Softw. Syst. Model.*, 15(1), 2013.
30. The Linux Foundation. Janusgraph, 2018.
31. Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Softw. Syst. Model.*, 2016.
32. Dániel Varró, Oszkár Semeráth, Gábor Szárnyas, and Ákos Horváth. *Towards the Automated Generation of Consistent, Diverse, Scalable and Realistic Graph Models*. Number 10800. 2018.
33. Gergely Varró, Frederik Deckwerth, Martin Wieber, and Andy Schürr. An algorithm for generating model-sensitive search plans for pattern matching on EMF models. *Softw. Syst. Model.*, pages 597–621, 2015.
34. Thomas Vogel and Holger Giese. Model-driven engineering of self-adaptive software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 8(4):18, 2014.
35. András Vörös, Márton Búr, István Ráth, Ákos Horváth, Zoltán Micskei, László Balogh, Bálint Hegyi, Benedek Horváth, Zsolt Mázló, and Dániel Varró. 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)*, volume 10811 LNCS, pages 460–467, 2018.
36. David S. Warren. Memoing for logic programs. *Commun. ACM*, 35(3):93–111, 1992.
37. Ben Zhang, Nitesh Mor, John Kolb, Douglas S Chan, Ken Lutz, Eric Allman, John Wawrzynek, Edward A Lee, and John Kubiatowicz. The cloud is not enough: Saving IoT from the cloud. In *7th USENIX Workshop on Hot Topics in Cloud Computing*, 2015.