2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)

Integrating Testing with Runtime Verification for Mission-Critical Distributed Control Systems

Davide Ancona*, Stefano Avola*, Angelo Ferrando†, Pierpaolo Baglietto*, Maurice H. ter Beek‡
Andrea Parodi§, Giancarlo Camera¶ and Matteo Pinasco¶
*DIBRIS – University of Genova, Genova, Italy
†University of Modena and Reggio Emilia, Modena, Italy
‡CNR–ISTI, Pisa, Italy, §M3S SrL, Genova, Italy, ¶Hitachi Rail STS SpA, Genova, Italy

***Abstract*—Verification of safety properties of mission-critical Distributed Control Systems (DCS) is challenging, especially when depending on a dynamically varying number of distributed components interacting via the system's Integration Layer (IL). In such cases, complementing testing with Runtime Verification (RV) can help detect non-systematic errors earlier and reduce time-to-production. We adopt RV to test the IL of a real-world mission-critical railway control system, based on a Message-oriented Middleware (MoM) implementing a publish-subscribe communication protocol, with critical requirements on message uniqueness and order. These requirements are formalized in RML (Runtime Monitoring Language) and compiled into a monitor which verifies them dynamically. Performance measurements on real-world scenario parameters show that our approach can complement testing in the Continuous Integration (CI) cycle.**

I. Introduction

Distributed systems are difficult to verify [31] because of their complexity, and non-deterministic behavior. This becomes more challenging when the System Under Scrutiny (SUS) is a mission-critical Distributed Control System (DCS), where validation is of paramount importance if incorrect behavior may seriously endanger human lives or infrastructures.

A typical major problem in a DCS is the correct integration of its different heterogeneous components through an Integration Layer (IL). At the core of the DCS, the IL code is subject to frequent revisions due to bug fixes, improvements, or changes in hardware or standards. For safety reasons, revisions of the IL require non-regression testing in the CI pipeline which, however, cannot verify properties involving several components interacting through the IL. Indeed, since the behavior of the SUS is highly non-deterministic and depends on a dynamically varying number of distributed components, most of the times critical safety functions of the SUS can be effectively tested only on the field, by means of costly system testing. Though this approach is followed to minimize the development time, finding critical bugs at the last stage might have the opposite effect.

In this case, formal verification techniques like model checking fail, due to the complexity of the IL and of the verified properties.

We investigate Runtime Verification (RV) [15] as a solution to complement testing, to avoid postponing to the system testing stage the checks of complex properties which cannot be verified with testing in the CI cycle. The beneficial effects are earlier detection of non-systematic errors with a consequent reduction of both the probability of failures of safety functions and of the time-to-production.

RV is a dynamic verification technique which checks the event trace generated by a single run of the SUS with monitors compiled from the specifications that define the correct behavior of the SUS, which is instrumented to generate the trace of events to be monitored. RV is complementary to formal verification and testing: like the former, it is based on specification formalisms, but it is not exhaustive [27]; like the latter, it scales well to real-world systems and complex properties. Differently from testing, RV is able to detect errors in non-deterministic SUS [5], [24], [32], [33], and, thus, to successfully complement testing [11].

Our investigation is based on a real-world railway control system developed by Hitachi Railway Signaling and Transportation Systems (Hitachi Railway STS), a mission-critical DCS with a complex IL subject to the rigorous safety and security constraints of the Safety Integrity Levels[1] (SIL) adopted in railway industry [12], [19].

The IL consists of a Message-oriented Middleware (MoM) implementing a publish-subscribe protocol [18], [20], [29]. Components communicate through the MoM by publishing messages and subscribing to specific topics. Different publishers are allowed to publish on the same topic and a publisher is allowed to publish different messages on different topics. Correspondingly, a subscriber is allowed to subscribe to different topics. If a publisher publishes a message m on topic t, then m is delivered by the MoM only to those components that are subscribed to t at the time m is published.

The correct implementation of the MoM implies that the following two mission-critical *properties* are satisfied for all publishers p, subscribers s, and topics t:

(exactly once): as long as s is subscribed to t, all messages published by p on t are delivered to s *exactly once*;

(same order): as long as s is subscribed to t, it receives the messages published by p on t in the *same order* as they have been sent. If a publisher p_2 publishes m_2 on t after p_1 has published m_1 on t, then a subscriber to t is allowed to receive m_2 before m_1 as long as $p_1 \neq p_2$.

[1] SIL denotes a standardized quantitative measure of the degree of reliability of safety functions in safety-critical systems, not to be confused with IL, abbreviation of Integration Layer.

2833-292X/25/$31.00 ©2025 IEEE
DOI 10.1109/DSN-S65789.2025.00056

The verification of these two properties consists in requiring that the set of event traces involving the communication acts between a publisher p and a subscriber s through topic t describes the correct behavior of a FIFO queue; such a set is not a context-free language [6]. Furthermore, such properties depend on a dynamically varying number of components.

Therefore, the adopted RV tool must be able to face the following *challenges*: **(i)** to be expressive to cater for the verification of **non-context-free** properties; **(ii)** to verify properties depending on **dynamically changing** parameters.

To this aim, we use the Runtime Monitoring Language (RML) [6], [21], a Domain-Specific Language (DSL) which has been used successfully for RV of interaction protocols in multi-agent systems [1], [2], [4], [14].

RML is expressive enough to tackle both of the above challenges **(i)** and **(ii)**. The instrumentation of the SUS is nontrivial since must ensure a synchronous interaction between the SUS and the monitor compiled from the RML specification.

Experiments with real-world parameters show that the average time required for RV to verify a single event is linear in the maximum number of publishers and subscribers, and that in a CI pipeline cycle about 784k communication events can be verified to detect non-systematic error due to the violation of the two properties **(exactly once)** and **(same order)**.

Related Work: Publish-subscribe communication protocols have the advantage of fostering high decoupling among components, yet they are difficult to reason about and to test [28].

We are not aware of any attempt in the literature to use RV to verify the implementation of a publish-subscribe protocol, although RV has recently attracted attention in the Internet of Things (IoT) [23], [26], [35], where such protocols are widely used. We are aware of only very few studies that propose RV for verifying systems based on a publish-subscribe protocol, and these typically concern verifying the properties of the system rather than the implementation itself [10], [23].

Decker et al. [16] propose RV to verify the communication of medical devices in service-oriented architectures. The proposed solution requires both a centralized and a local monitor, but no publish-subscribe protocol is used for communication.

The literature contains many proposals based on model checking [8], [9], [17], [22], [30], [34] to verify properties of systems whose components interact via the publish-subscribe protocol. However, in contrast to our aim of verifying a real implementation, these studies establish properties of a model. Furthermore, most model checking-based approaches neglect many characteristics of the underlying publish-subscribe communication to avoid state-space explosion problems [8].

Outline: Section II explains the Hitachi Railway STS MoM. Section III briefly introduces RML and Section IV details the RML specification of the verified properties. Section V shows the instrumentation of the MoM, after which Section VI reports on the experiments. Section VII concludes with lessons learned and directions for future development. The related data and code are available from an OSF repository [7][2].

[2] A non-disclosure agreement prevents us to publish the code of the MoM.

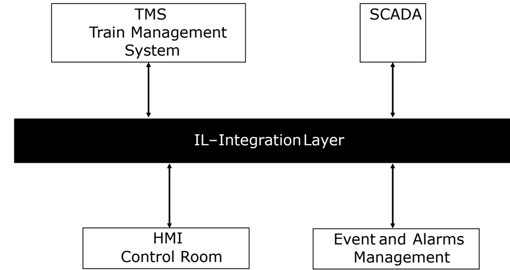

Fig. 1. General overview of the Integration Layer (IL)

II. Overview of the MoM

The MoM subject to RV is the core component of the IL of a mission-critical railway control system architecture developed by the company Hitachi Railway STS. The IL component (see Figure 1) serves as the central integration solution for the many technologies developed as part of the full Rail Operations Center (ROC) systems. The IL is a single access point for the full state of the railway network, and provides an efficient solution for several integration requirements and patterns [25], where the publish-subscribe pattern plays a crucial role.

The railway control system components in Figure 1 connect to the IL either as source of information (publishers), as destination of information (subscribers), or as both. The data that the components in Figure 1 publish or read on specific topics include the following non-exhaustive list.

TMS (Train Management System): the control system's core component. It subscribes to receive i) the full state of the railway network (i.e., train position sensors, signaling, switches) and ii) any alarm generated by the other components; it publishes a lot of data (e.g., trains' delays).

SCADA (Supervisory Control and Data Acquisition): component with the role of supervising all the technical equipment (e.g, the power line); it mainly publishes prompts when anomalies are detected.

EMS (Event and Alarms Management System): subscribes to receive all anomalies prompts; by means of several proprietary algorithms it can identify critical situations (alarms) and then it publishes to the relevant topics.

HMI (Human Machine Interface): control system's frontend used by the traffic controllers. It provides the railway line layout visually representing all system info (e.g., train positions, switches state). It subscribes to all info published by other components (e.g., alarms, network state); it publishes command events directed at controlling the underlying equipment.

The IL is subject to many rigorous safety and security constraints, such as the SIL [12], [19]. The assessment of these preconditions for the production-level operations of the component includes correctness properties like **(exactly once)** and **(correct order)**, as defined in Section I. For instance, if the TMS misses a message published on topic `'HMIcmd'` by the HMI to activate a switch, then quite hazardous situations might occur, and this would prevent the ROC to pass even the lowest level of safety (i.e., SIL 1).

III. An Introduction to RML

RML [6], [21] is a rewriting-based and system-agnostic DSL for RV, enabling developers to define formal specifications independently of code instrumentation. It builds upon prior work in RV and global types, with applications in various contexts, including the verification of interaction protocols in multi-agent systems [1], [2], [4], [14].

The pillar concepts of RML are **event types** (sets of events) and **trace expressions** (sets of event traces).

The language supports parametric and generic specifications, and its expressive power allows verification of non-context-free properties—a capability not readily achievable with other commonly adopted formalisms for RV, like those based on regular expressions [3], context-free grammars or linear temporal logic.

RML specifications are compiled into monitors which verify the corresponding properties while the SUS is running.

RML *events* are represented with object literals[3] specifying their properties. For instance, the following object represents the event 'publisher with id 9 has published message with id 7 on the topic `'HMIcmd'`':

```
{agent:'pub',op:'send',id:9,topic:'HMIcmd',msgId:7}
```

A specification defines the set of correct event traces of the runs of the SUS; the monitor compiled from it checks that the trace of a single run of the SUS belongs to such a set.

Specifications are built on top of event *patterns* and *types*. Event types can be parametric, like the event type `pub` below with three parameters, and matching events corresponding to the publication on `topic` by the publisher with id `pubId` of the message with id `msgId`.

```
pub(pubId,topic,msgId) matches
{agent:'pub', op:'send', id:pubId,topic:topic,msgId:msgId};
```

The event pattern `pub(id,'HMIcmd',_)` matches all publications on topic `'HMIcmd'`. If the match is successful, then the variable `id` is instantiated with the corresponding publisher id; the wildcard _ specifies that the information on the id of the published message is not relevant for the match.

In RML derived event types can be defined:

```
notSubs not matches subs;
subsOrRecv(subId,topic) matches subs(subId,topic) |
    recv(subId,topic,_,_);
```

The event pattern `notSubs` matches all events which do not match a subscription (event type `subs`); `subsOrRecv(id,t)` matches all events matching either `subs(id,t)` or `recv(id,t,_,_)`.

RML is a trace-based specification language [13]: trace expressions together with recursion define sets of event traces on top of type patterns and primitive and derived operators.

Primitive operators include the constant `empty` (the singleton set with the empty trace), *concatenation* $t_1\ t_2$, *intersection* $t_1 \wedge t_2$, *union* $t_1 \vee t_2$, and *shuffle* $t_1 \mid t_2$.

Derived operators include the standard regular expression operators `?`, `+`, and `*`, the constant `all`, representing the universe of all traces, and the conditional filter $\vartheta \gg t_1 : t_2$, which denotes the set of all traces verifying t_1 when restricted to the events matching ϑ, and t_2 when restricted to the events not matching ϑ; $\vartheta \gg t$ is an abbreviation for $\vartheta \gg t$: `all`.

The following specification defines the property that publisher with id 0 can publish messages only after its creation:

```
pub(pubId) matches pub(pubId,_,_);
newPub(pubId) matches {agent:'pub',op:'new',id:pubId};
NewPubThenPub0 = newPub(0) pub(0)*;
```

Specifications can be generic, i.e., `NewPubThenPub0` can be abstracted:

```
NewPubThenPub<pubId> = newPub(pubId) pub(pubId)*;
```

By existentially quantifying variable `pub`, `NewPubThenPub0` above can be parametric in an arbitrary number of publishers:

```
NewPubThenPub = {let pubId; newPub(pubId) (pub(pubId)* |
    NewPubThenPub )}?;
```

This is achieved in conjunction with recursion, the shuffle operator, and the `let` construct. For instance, if publishers 0 and 1 are created, then `NewPubThenPub` rewrites to `pub(0)*` | `pub(1)*` | `NewPubThenPub`. The shuffle operator allows events matching `pub(0)` or `pub(1)` to occur in any order.

The operators of RML together with recursion and parametricity allow the definition of non-trivial properties. The specification below fully captures[4] the properties **(exactly once)** and **(same order)** of the MoM, i.e., correct behavior of FIFO queues, whose corresponding set of traces is a non-context-free language [6]:

```
Queue = {let val; enq(val) ((deq | Queue) ∧ (deq >>
    deq(val) all))}?;
```

Pattern `deq` matches any dequeuing, while `enq(val)` and `deq(val)` match enqueuing and dequeuing of value `val`, resp.

The expression `(deq | Queue)` states that after an enqueue, a dequeue event is allowed to eventually occur. The expression `(deq ≫ deq(val) all)` corresponds to the constraint "if a value is dequeued, then it must be equal to `val`". Since the constraint is in conjunction with `(deq | Queue)`, the constant `all` follows, to specify that after the value is dequeued any other event can occur, i.e., no further constraint needs to be verified. The final `?` allows the empty trace, corresponding to no activity on the queue.

IV. Specification

Before defining all needed trace expressions, the specification introduces all the required event types.

A. Event Types

Figure 2 defines the specification's main event types. Event type `pub(pubId, topic, msgId)` was introduced in Section III. Subscriptions on `topic` by subscriber `subId` are matched by the event type `subs(subId, topic)`; `recv(subId, topic, msgId, pubId)` defines events corresponding to subscriber `subId` receiving message with id `msgId` published on `topic` by publisher `pubId`; and

[3] Serialized and deserialized in JSON format.

[4] Red and blue specify **(exactly once)** and **(same order)**, respectively.

```
pub(pubId, topic, msgId) matches {agent: 'pub', op:
    'send', id: pubId, topic: topic, msgId: msgId};
subs(subId, topic) matches {agent: 'sub', op:
    'subscription', id: subId, topic: topic};
recv(subId, topic, msgId, pubId) matches {agent:
    'sub', op: 'receive', id: subId, topic: topic,
    msgId: msgId, sender: pubId};
newPub(pubId) matches {agent: 'pub', op:'new',
    id:pubId};
// derived event types omitted
```

Fig. 2. Main event types of the specification

```
Main = relevant ≫
  (
   CheckSubs  ∧ CheckPub
   NoMultipleSubs  ∧ NoMultipleNewPub  ∧
   (subsOrRecv  ≫ SubsThenRecv)  ∧
   (newPubOrPub  ≫ NewPubThenPub)  ∧
  );
Queue<pubId, subId, topic> =  let msgId; pub(pubId,
    topic, msgId) ((recv | Queue<pubId, subId,
    topic>) ∧ (recv >> recv(subId, topic, msgId,
    pubId) all))?;
CheckSubs = notSubs* let subId, topic; subs(subId,
    topic) (GenCheckSubs<subId, topic> ∧
    CheckSubs)?;
GenCheckSubs<subId, topic> = notNewPub* let pubId;
    newPub(pubId) ((involve(pubId, subId, topic) >>
    Queue<pubId, subId, topic>) ∧
    GenCheckSubs<subId, topic>)?;
NoMultipleSubs = notSubs* {let topic, subId;
    subs(subId, topic)(notSubs(subId,topic)*  ∧
    NoMultipleSubs)}?;
SubsThenRecv = {let topic, subId; subs(subId,
    topic) (recv(subId,topic)* | SubsThenRecv )}?;
// similar definitions emitted
```

Fig. 3. Selected code from the specification

`newPub(pubId)` matches creation of a publisher `pubId`. The monitor needs the latter to keep track of all publishers and to verify that all subscribers receive all their messages correctly. The remaining event types (omitted in the figure) are derived and only instrumental to the specification.

B. Trace expressions

Figure 3 shows a selection of the defined trace expressions. The main one is the conjunction of six expressions, restricted to the events relevant for the verification, i.e., those matching the events in Figure 2. The generic specification `Queue<pubId,subId,topic>` (highlighted in red) formalizes the properties **(exactly once)** and **(same order)** (challenge **(i)** from Section I), by adopting the pattern for FIFO queues of Section III: event types `pub(pubId, topic, msgId)` and `recv(subId, topic, msgId, pubId)` correspond to `enq(msgId)` and `deq(msgId)`, resp. All messages published by `pubId` on `topic` will be received by `subId` *exactly once* and in the *same order* as happens in a correct implementation of a queue. To this aim, the information `msgId` is essential to distinguish the different messages. The three parameters of `Queue` reflect the fact that the property is verified for all possible dynamically changing triples consisting of a publisher `pubId`, a subscriber `subId`, and a topic `topic`.

Challenge **(ii)** of Section I is handled by `CheckSubs` (highlighted in blue) and `CheckPub`, which indirectly depend on `Queue<pubId,subId,topic>`, and verify events after a subscription or a creation of a publisher occurs, resp. Both specifications are needed because `CheckSubs` deals with subscriptions occurring *before* publisher creation, while `CheckPub` deals with the other way around. For symmetry, only the details on `CheckSubs` are provided.

In `CheckSubs`, if `subId` subscribes to `topic`, then `GenCheckSubs<subId,topic>` verifies (see below) all publications on `topic`, receptions by `subId` on `topic`, and creations of a publisher. The definition of `CheckSubs` is recursive since for each subscription its corresponding property must be verified in conjunction with the others. The expression `notSubs*` specifies that the initial events of the traces of `CheckSubs` can be ignored as long as they are not subscriptions.

In `GenCheckSubs<subId,topic>`, if a publisher `pubId` is created, then `Queue<pubId,subId,topic>` verifies all messages published by `pubId` on `topic` and received by `subId`; these are the only events kept with the filter operator `involve(pubId,subId,topic)`≫_. Recursion is needed for the same reasons for `CheckSubs`.

The remaining definitions shown in Figure 3 specify simpler properties which however are expected to hold for a correct implementation of the MoM.

`NoMultipleSubs` forbids multiple subscriptions to the same topic by the same subscriber. An analogous specification `NoMultipleNewPub` (not shown in Figure 3) checks the same property for publisher creation. Since multiple occurrences of the same subscription and creation of the same publisher can never occur, verification of these two subspecifications ensures that `CheckSubs` and `CheckPub` actually check disjoint events.

`SubsThenRecv` verifies that subscribers cannot receive messages on a topic before subscribing to it. `NewPubThenPub` (not shown in Figure 3) verifies the same property for publisher creation and message publication.

V. Instrumentation

The aim of the instrumentation of the SUS is twofold: to send to the monitor the trace of relevant events, and to collect measurements needed to assess the performances of the approach in Section VI.

To verify temporal global properties such as **(same order)** of Section I, the instrumentation must ensure an asynchronous communication between the SUS and the monitor. For instance, if the MoM receives two messages published by the same publisher on the same topic, then the corresponding events must be received by the monitor in the same order as by the MoM.

To this aim, the instrumentation[5] is based on the architecture in Figure 4. The monitor is a WebSocket server which receives events of the SUS in JSON format, and sends back the corresponding verdict. Publishers and subscribers (referred to below as "agents") do not interact directly with the MoM, but

[5]The instrumentation required roughly 1.2 k SLOC.

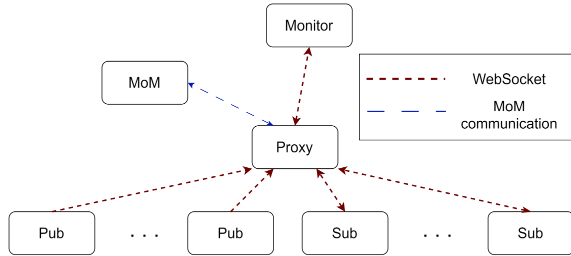

Fig. 4. Overall architecture of the instrumentation

communicate with a *proxy* by using the WebSocket protocol. A verdict carries a Boolean value, to notify whether the specification has been violated, together with the monitored event. Since our aim is to use RV only during testing, no action recovery from errors is ever attempted.

The proxy is implemented as an N-client to the MoM, where N is the total number of monitored agents. For each monitored agent, it calls a corresponding handler to manage the communications between the agent and the MoM.

In the first phase, when a new publisher `p` is created, it notifies the proxy through the message queue used by the proxy. Then, in the subsequent phase, a corresponding handler is created for the new publisher and an event of type `newPub(p)` is sent to the monitor. After the monitor's response is received, the handler can connect to the MoM.

When a publisher `p` asks the proxy to publish message `m` on topic `t`, the corresponding event of type `pub(p, t, m)` is sent by the proxy to the monitor. After the monitor's response is received, the proxy forwards `m` to the corresponding handler which, eventually, publishes `m` on topic `t` by using the MoM.

For subscriptions, the communication flow is similar, except for the event of type `subs(s, t)` is sent by the proxy to the monitor, where `s` identifies the subscriber and `t` is the topic.

As depicted in Figure 4, differently from the monitored publishers, a two-way communication is established between the monitored subscribers and the proxy: when a handler of a subscriber `s` receives from the MoM a message `m` on topic `t`, the proxy sends the corresponding event of type `recv(s, t, m, p)` to the monitor. After the monitor's response is received, the proxy forwards the message to the monitored subscriber.

The proxy handles all messages coming from the agents with a single queue to ensure that all interactions are synchronous and, thus, the order of the monitored events is the same as the order of the corresponding ones received and verified by the monitor. In particular, every time a handler sends a message to the MoM, the next element of the queue is processed only after the handler has received the corresponding acknowledge message by the MoM.

Three different levels of instrumentation are supported: (0) no instrumentation: agents directly interact with the MoM; (1) partial instrumentation: agents interact through the proxy, but the monitor is not used; (2) full instrumentation: both the proxy and the monitor interact with the SUS.

The instrumentation measures the number of events processed by the monitor, and the overall time needed by the (instrumented) SUS to complete the corresponding actions: creation of a publisher, subscription to a topic, and publication and reception of a message. For instrumentation level 0, the measurement for the first three types of events corresponds to the time elapsed between the sending of the request to the MoM and the reception of the corresponding acknowledge message. The measurement for the reception of messages by a subscriber is the time elapsed after the publishing request was sent to the MoM.

For the other instrumentation levels, the measured time includes also the extra time needed by the communication between the SUS and the proxy (levels 1 and 2) and the proxy and the monitor (level 2 only), and the time needed by the monitor to process the received events (level 2 only).

VI. Experiments

To measure the performance overhead, the monitor, proxy, publishers, and subscribers were run on a physical Windows machine (4-core CPU, 16 GB RAM), while the MoM was run separately over Kubernetes on 4 Linux virtual machines (4-core CPU, 8 GB RAM each) running on a different bare metal server. Network latency between the two was less than 1 ms.

In the experiments, on-the-field numbers of topics, publishers, and subscribers were taken from a real deployment of a high-speed railway control system.

Since the number of publications and receptions of message events in this scenario is typically several orders of magnitude larger than the number of creations of publishers and subscriptions, we considered only the events for publication and reception of messages together with their corresponding times to measure the overhead. The measurements were conducted at the different instrumentation levels defined in Section V, and by varying the number of publishers and subscribers.

A test collects the accumulative times for all publications (abbreviated as "pub") and receptions by subscribers ("sub").

To obtain more robust values, the interquartile range[6] was computed to remove outliers. After that, the median of the remaining values was considered. Below, we simply use the term "measured time" to mean such a value.

The total elapsed time for each run of a test was 15 minutes. This means that for each test the runs of the SUS were effectively monitored for about 5 minutes, since the remaining time was used by the initialization and ending phases.

Every test was run three times and the average of the measured times was picked. Since the standard deviation of the measured times was about 10%, we decided that three times was a reasonable choice.

For each experiment, the following most relevant topics were chosen: one concerned with data coming from SCADA systems, one with data generated by the event and alarm management systems, and the last one with data sent by the TMS and shown by the HMI (see Figure 1).

[6] The interquartile range *IQR* is defined as the difference between the 75th-percentile Q_3 and the 25th-percentile Q_1.

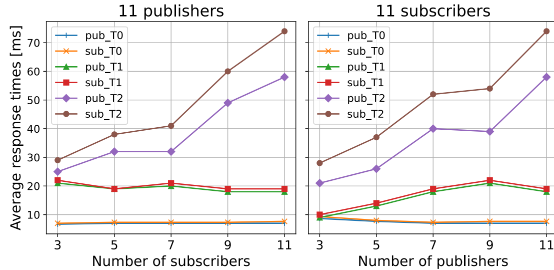

Fig. 5. Fixed number of publishers (left) and subscribers (right)

Figure 5 contains two plots with the results of the experiments: on the left-hand side, performance is plotted where the number of publishers was kept constantly equal to 11, while the number of subscribers varied from 3 to 11. The plot on the right-hand side depicts the dual scenario.

The different average measured times (expressed in ms) are reported per type of event and instrumentation level.

To distinguish the times for different instrumentation levels (defined in Section V), we used T_k, where $k = \{0, 1, 2\}$. Thus in the 11 subscribers plot, with 9 publishers the average response time for publishing messages with instrumentation level 2 is ± 40 ms, but less than 10 ms with no instrumentation.

As expected, the overhead increases linearly with the number of agents, with some fluctuations which cannot easily be predicted, because of the complexity of the overall system. For instance, the automatic garbage collection process both at the level of the monitor and the agents and proxy, can cause some instability in the results on the measurements trend.

The highest average response times are measured for the receptions of messages by subscribers with instrumentation level 2. Such a time does not exceed 75 ms.

For the largest experiment (11 publishers + 11 subscribers) with level 2, about 30 k events were generated and monitored. To further test the approach's robustness we ran all tests with level 2 and within the much larger time frame[7] of 90 minutes, with a total number of about 784 k monitored events.

VII. Conclusion

We employed RV to verify properties of a mission-critical DCS which cannot be checked by non-regression testing.

We verified the correctness of the publish-subscribe protocol as implemented in a railway control system developed by Hitachi Railway STS, to ensure the uniqueness **(exactly once)** and correct order **(same order)** of the exchanged messages. To this aim, we tackled two main challenges: the properties are **(i) non-context-free** and depend on **(ii) dynamically changing** parameters. Such properties have been defined in RML and compiled into a monitor able to dynamically verify them. To verify **(same order)**, the instrumentation required a proxy agent to ensure a synchronous communication between all agents and the monitor.

[7]Typically, this time compares to the duration of the non-regression testing pipeline required for the continuous integration of the railway control system.

We assessed our approach by measuring the execution overhead for parameters taken from requirements for real on-the-field deployment of a high-speed railway control system.

Lesson Learned: We can draw both positive and negative conclusions. On the positive side, the experiments showed that the proposed approach is scalable and can be effectively integrated in the CI pipeline, where many unit and integration tests are executed at each development iteration, for the purpose of non-regression testing. Typically, a complete execution of the pipeline takes at least 90 minutes; we have measured that within this time the monitor is able to verify about 784 k communication events to detect non-systematic errors of the MoM of the IL which **cannot** be checked by the pipeline tests.

The use of RML turned out to be a winning choice, although other formalisms could have been considered as well, but with hard constraints on the expressive power and on the ability to handle highly parametric specifications. The first constraint alone is sufficient to prevent the use of several temporal logics (like LTL) not able to express non-context-free properties [3] like those checked in our case study. Furthermore, RML allowed to write an agnostic specification which can be reused to verify the same properties in DCSs where similar protocols are adopted, like the widespread MQTT protocol.

On the negative side, while the solution can be applied in the CI cycles, the performances show that its usage is prohibitive when monitoring a mission-critical DCS on the field.

Also, instrumentation efforts needed to ensure synchronous communication should not be underestimated when considering to follow this approach. However, the net gain is that, once the infrastructure is implemented, it can be reused to verify further properties of the DCS.

Future Developments: To integrate our solution into the CI pipeline, more experiments are needed to estimate which are the optimal duration and frequency of RV for a good balance between the found bugs and the consumed resources. Since the tests in the CI pipeline are not able to detect violations of the properties checked with RV, our solution can only improve the rate of found bugs, but such a rate could be increased with a tool for distributed per-test coverage to ensure that the SUS trace covers the majority of the verified code.

Acknowledgments: Supported by the MUR PRIN 2020TL3X8X project T-LADIES (Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems) and the MUR PRIN 2022 PNRR P2022A492B project ADVENTURE (ADVancEd iNtegraTed evalUation of Railway systEms) and the MOST – Sustainable Mobility National Research Center and received funding from the European Union Next-GenerationEU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) – MISSIONE 4, COMPONENTE 2, INVESTIMENTO 1.4 – D.D. 1033 17/06/2022, CN00000023. Funded by the European Union - NextGenerationEU and by the Ministry of University and Research (MUR), National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.5, project "RAISE - Robotics and AI for Socio-economic Empowerment" (ECS00000035). This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

REFERENCES

[1] Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P., Gay, S.J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E.B., Martins, F., Mascardi, V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V.T., Yoshida, N.: Behavioral types in programming languages. Found. Trends Program. Lang. **3**(2-3), 95–230 (2016). doi:10.1561/2500000031

[2] Ancona, D., Drossopoulou, S., Mascardi, V.: Automatic Generation of Self-monitoring MASs from Multiparty Global Session Types in Jason. In: Baldoni, M., Dennis, L.A., Mascardi, V., Vasconcelos, W.W. (eds.) Revised Selected Papers of the 10th International Workshop on Declarative Agent Languages and Technologies (DALT'12). LNCS, vol. 7784, pp. 76–95. Springer (2012). doi:10.1007/978-3-642-37890-4_5

[3] Ancona, D., Ferrando, A., Mascardi, V.: Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification. In: Ábrahám, E., Bonsangue, M.M., Johnsen, E.B. (eds.) Theory and Practice of Formal Methods. LNCS, vol. 9660, pp. 47–64. Springer (2016). doi:10.1007/978-3-319-30734-3_6

[4] Ancona, D., Ferrando, A., Mascardi, V.: Parametric Runtime Verification of Multiagent Systems. In: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems (AAMAS'17). pp. 1457–1459. ACM (2017). doi:10.5555/3091125.3091328

[5] Ancona, D., Franceschini, L., Delzanno, G., Leotta, M., Ribaudo, M., Ricca, F.: Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things. In: Pianini, D., Salvaneschi, G. (eds.) Proceedings of the 1st Workshop on Architectures, Languages and Paradigms for IoT (ALP4IoT'17). EPTCS, vol. 264, pp. 27–42 (2017). doi:10.4204/EPTCS.264.4

[6] Ancona, D., Franceschini, L., Ferrando, A., Mascardi, V.: RML: Theory and practice of a domain specific language for runtime verification. Sci. Comput. Program. **205**, 102610 (2021). doi:10.1016/J.SCICO.2021.102610

[7] OSF repository with data and code associated with the submitted paper, https://osf.io/2c5rh/?view_only=f61f5501890f43f8b4f72bb095a2e810

[8] Baresi, L., Ghezzi, C., Mottola, L.: Loupe: Verifying Publish-Subscribe Architectures with a Magnifying Lens. IEEE Trans. Softw. Eng. **37**(2), 228–246 (2011). doi:10.1109/TSE.2010.39

[9] ter Beek, M.H., Gnesi, S., Latella, D., Massink, M., Sebastianis, M., Trentanni, G.: Assisting the design of a groupware system – Model checking usability aspects of thinkteam. J. Log. Algebr. Program. **78**(4), 191–232 (2009). doi:10.1016/j.jlap.2008.11.004

[10] Begemann, M.J., Kallwies, H., Leucker, M., Schmitz, M.: TeSSLa-ROS-Bridge – Runtime Verification of Robotic Systems. In: Ábrahám, E., Dubslaff, C., Tapia Tarifa, S.L. (eds.) Proceedings of the 20th International Colloquium on Theoretical Aspects of Computing (ICTAC'23). LNCS, vol. 14446, pp. 388–398. Springer (2023). doi:10.1007/978-3-031-47963-2_23

[11] Besnard, V., Huet, M., Bivolarov, S., Saadi, N., Cornard, G.: AMT: A Runtime Verification Tool of Video Streams. In: Katsaros, P., Nenzi, L. (eds.) Proceedings of the 23rd International Conference on Runtime Verification (RV'23). LNCS, vol. 14245, pp. 315–326. Springer (2023). doi:10.1007/978-3-031-44267-4_16

[12] Boulanger, J.L.: CENELEC 50128 and IEC 62279 Standards. Wiley (2015)

[13] Bubel, R., Gurov, D., Hähnle, R., Scaletta, M.: Trace-based Deductive Verification. In: Piskac, R., Voronkov, A. (eds.) Proceedings of the 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'23). EPiC Series in Computing, vol. 94, pp. 73–95. EasyChair (2023). doi:10.29007/VDFD

[14] Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On Global Types and Multi-Party Sessions. Log. Methods Comput. Sci. **8**(1:24), 1–45 (2012). doi:10.2168/LMCS-8(1:24)2012

[15] Colombo, C., Pace, G.J.: Runtime Verification: A Hands-On Approach in Java. Springer (2022). doi:10.1007/978-3-031-09268-8

[16] Decker, N., Kühn, F., Thoma, D.: Runtime Verification of Web Services for Interconnected Medical Devices. In: Proceedings of the 25th International Symposium on Software Reliability Engineering (ISSRE'14). pp. 235–244. IEEE (2014). doi:10.1109/ISSRE.2014.16

[17] Deng, X., Dwyer, M.B., Hatcliff, J., Jung, G., Robby, Singh, G.: Model-Checking Middleware-Based Event-Driven Real-Time Embedded Software. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Revised Lectures of the 1st International Symposium on Formal Methods for Components and Objects (FMCO'02). LNCS, vol. 2852, pp. 154–181. Springer (2002). doi:10.1007/978-3-540-39656-7_6

[18] Eugster, P.T., Felber, P., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2), 114–131 (2003). doi:10.1145/857076.857078

[19] European Committee for Electrotechnical Standardization: CENELEC EN 50128: Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems (June 2011), https://standards.globalspec.com/std/1678027/cenelec-en-50128

[20] Fiege, L., Cilia, M., Mühl, G., Buchmann, A.P.: Publish-Subscribe Grows Up: Support for Management, Visibility Control, and Heterogeneity. IEEE Internet Comput. **10**(1), 48–55 (2006). doi:10.1109/MIC.2006.17

[21] Franceschini, L.: RML: Runtime Monitoring Language. Ph.D. thesis, DIBRIS – University of Genova (March 2020), http://hdl.handle.net/11567/1001856

[22] Garlan, D., Khersonsky, S., Kim, J.S.: Model Checking Publish-Subscribe Systems. In: Ball, T., Rajamani, S.K. (eds.) Proceedings of the 10th International SPIN Workshop on Model Checking Software (SPIN'03). LNCS, vol. 2648, pp. 166–180. Springer (2003). doi:10.1007/3-540-44829-2_11

[23] Grochowski, M., Kowalewski, S., Buchsbaum, M., Brecher, C.: Applying Runtime Monitoring to the Industrial Internet of Things. In: Proceedings of the 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'19). pp. 348–355. IEEE (2019). doi:10.1109/ETFA.2019.8869447

[24] Havelund, K., Roşu, G.: An Overview of the Runtime Verification Tool Java PathExplorer. Form. Methods Syst. Des. **24**(2), 189–215 (2004). doi:10.1023/B:FORM.0000017721.39909.4B

[25] Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison-Wesley (2004)

[26] Inçki, K., Ari, I.: A Novel Runtime Verification Solution for IoT Systems. IEEE Access **6**, 13501–13512 (2018). doi:10.1109/ACCESS.2018.2813887

[27] Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Methods Program. **78**(5), 293–303 (2009). doi:10.1016/j.jlap.2008.08.004

[28] Michlmayr, A., Fenkam, P., Dustdar, S.: Architecting a Testing Framework for Publish/Subscribe Applications. In: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06). pp. 467–474. IEEE (2006). doi:10.1109/COMPSAC.2006.28

[29] Mühl, G., Fiege, L., Pietzuch, P.R.: Distributed Event-Based Systems. Springer (2006). doi:10.1007/3-540-32653-7

[30] Rodríguez, A., Kristensen, L.M., Rutle, A.: Verification of the MQTT IoT Protocol Using Property-Specific CTL Sweep-Line Algorithms. In: Koutny, M., Kordon, F., Pomello, L. (eds.) Transactions on Petri Nets and Other Models of Concurrency XV, LNCS, vol. 12530, pp. 165–183. Springer (2021). doi:10.1007/978-3-662-63079-2_8

[31] Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bianculli, D., Colombo, C., Falcone, Y., Francalanza, A., Krstic, S., Lourenço, J.M., Ničković, D., Pace, G.J., Rufino, J., Signoles, J., Traytel, D., Weiss, A.: A survey of challenges for runtime verification from advanced application domains (beyond software). Form. Methods Syst. Des. **54**(3), 279–335 (2019). doi:10.1007/S10703-019-00337-W

[32] Schiavio, F., Sun, H., Bonetta, D., Rosà, A., Binder, W.: NodeMOP: runtime verification for Node.js applications. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC'19). pp. 1794–1801. ACM (2019). doi:10.1145/3297280.3297456

[33] Sharma, S., Gopalakrishnan, G., Mercer, E., Holt, J.: MCC: A runtime verification tool for MCAPI user applications. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD'09). pp. 41–44. IEEE (2009). doi:10.1109/FMCAD.2009.5351145

[34] Tripakis, S., Yovine, S.: Timing Analysis and Code Generation of Vehicle Control Software using Taxys. In: Havelund, K., Rosu, G. (eds.) Proceedings of the 1st Workshop on Runtime Verification (RV'01). Electron. Notes Theor. Comput. Sci., vol. 55, pp. 277–286. Elsevier (2001). doi:10.1016/S1571-0661(04)00257-9

[35] Tsigkanos, C., Bersani, M.M., Frangoudis, P.A., Dustdar, S.: Edge-Based Runtime Verification for the Internet of Things. IEEE Trans. Serv. Comput. **15**(5), 2713–2727 (2022). doi:10.1109/TSC.2021.3074956