Visual Self-Healing Modelling for Reliable Internet-of-Things Systems

João Pedro Dias$^{1,2[0000-0001-9066-6436]}$, Bruno Lima$^{1,2[0000-0003-2572-047X]}$, João Pascoal Faria$^{1,2[0000-0003-3825-3954]}$, André Restivo$^{1,3[0000-0002-1328-3391]}$, and Hugo Sereno Ferreira$^{1,2[0000-0002-4963-3525]}$

[1] DEI, Faculty of Engineering, University of Porto, Porto, Portugal
[2] INESC TEC, Porto, Portugal
[3] LIACC, Porto, Portugal
{jpmdias,bruno.lima,jpf,arestivo,hugosf}@fe.up.pt

Abstract. Internet-of-Things systems are comprised of highly heterogeneous architectures, where different protocols, application stacks, integration services, and orchestration engines co-exist. As they permeate our everyday lives, more of them become safety-critical, increasing the need for making them testable and fault-tolerant, with minimal human intervention. In this paper, we present a set of self-healing extensions for Node-RED, a popular visual programming solution for IoT systems. These extensions add runtime verification mechanisms and self-healing capabilities via new reusable nodes, some of them leveraging *meta-programming* techniques. With them, we were able to implement self-modification of *flows*, empowering the system with self-monitoring and self-testing capabilities, that search for malfunctions, and take subsequent actions towards the maintenance of health and recovery. We tested these mechanisms on a set of scenarios using a live physical setup that we called *SmartLab*. Our results indicate that this approach can improve a system's reliability and dependability, both by being able to detect failing conditions, as well as reacting to them by self-modifying *flows*, or triggering countermeasures.

Keywords: Internet-of-Things · Runtime Verification · Self-Healing · Software Engineering · Visual Programming

1 Introduction

The Internet-of-Things (IoT) is a network of programmable uniquely identifiable devices, known as *things*, that can sense (*i.e.*, sensors) and change (*i.e.*, actuators) their environment [22]. Within the nature of IoT systems, there are several particularities that, although not new or unique, congregate at an unprecedented scale in terms of interconnected devices, people, systems, and information resources, leading to an ever-increasing complexity that developers must address. These systems — typically built with heterogeneous parts, mostly resulting from the integration of different, and, sometimes, already existent, systems (*i.e.*, systems of systems [8]) — are not only logically distributed but also geographically, and commonly have to deal with power constraints and real-time needs.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50426-7_27

The wide range of IoT application scenarios urges the need for tools that allow users with reduced technical knowledge to configure and adapt their systems to their needs. These requirements lead to the birth of several different *low-code* and visual programming solutions that try to reduce the inherent complexity of programming and configuring these systems. Ray et al. [29] identify several visual programming solutions tailored to this domain. To get a grasp on the popularity of these solutions, we surveyed open-source tools (hosted on GitHub), using the number of stars as the primary metric. We can observe that the most popular visual programming solution for this domain is, by far, Node-RED (9600 stars), followed by XOD (583), Ardublock (376, Arduino-only development), Snap4Arduino (99, Arduino-only), Wyliodrin (84), Intel IoT Services Orchestration Layer (80), miniBloq (72), and NETLabToolkit (17, Arduino-only).

As systems' complexity increases, it inevitably results in people becoming *"overwhelmed by the effort to properly control the assembled collection,"* [28] increasing the probability of human-induced errors and failures; developing becomes hard, labor-intensive, and expensive, no matter how *low-code* the infrastructure is [16]. IoT is acknowledged to be a particular example of these complex systems [23], where recovering from faults becomes challenging [14]. As a result of this inherent complexity, researchers have argued that there is an imminent need for *autonomic components* [4,3,31]. From single devices (*e.g.*, smart locks) to whole systems (*e.g.*, smart homes), components should be capable of self-management, reducing the need for frequent human interactions [18]. This becomes essential in mission-critical systems, or when devices are deployed in remote locations (*e.g.*, wildfire control) or hard to access areas (*e.g.*, inside walls).

Ganek and Corbi [13] identify four desired *self* properties, namely: *self-configuring*, *self-healing*, *self-optimization*, and *self-protection*. All these characteristics require a certain degree of *runtime introspection* [19] from the system. Monitoring has been the most common approach for understanding a running system [12,1,15]; this technique allows one to retrieve operational data about running systems by using several distinct methods, but it is usually done by external tools and without a *feedback loop*. Some authors do propose the usage of runtime verification as a way to detect malfunctions and failures of system elements and their interactions [1], which act as a *lightweight* verification mechanism, complementing techniques such as model checking and testing. The main difference lies in providing the missing *feedback loop*, allowing taking actions as soon as some incorrect behavior is detected [21]. This verification mechanism can be used as a foundation for self-healing IoT systems.

Our work focuses on the principle that systems should be able to reconfigure themselves to recover from failures introduced by faulty parts. To achieve this, the running system must be able to model itself so that it can identify the faulty components during its operation (*i.e.*, runtime), without the need for human inspection. Our main contribution is the ability to visually model diagnosis and recovery/maintenance of health mechanisms to improve IoT systems' reliability, thus enabling them to be *self-healing*. These mechanisms have been developed, applied, and tested, as extensions that we named Self-Healing Extensions for

Node-RED (SHEN). We validated our approach by executing a set of scenarios on top of a live, physical setup, called *SmartLab*. The in-place based system was first upgraded with the designed extensions; then, a set of common scenarios was executed, and the resulting system behavior observed. Our experiments show the feasibility of the approach, pointing to improvements in terms of system reliability and dependability, despite several limitations and challenges that this particular VPL language poses, and which limit the full potential of our approach.

The structure of the remaining paper is as follows: Section 2 provides an overview of the main concepts, Section 3 explores related work, Section 4 describes our approach for runtime verification and self-healing as Node-RED extensions, Section 5 describes the experimental phase, Section 6 discusses the limitations, challenges, and benefits of our approach and Node-RED itself, and lastly, Section 7 provides some final remarks.

2 Preliminaries

The following paragraphs introduce some fundamental building blocks and key concepts of this work, focusing on IoT. The Node-RED tool is presented (2.1) along with additional details about its functioning and known limitations. The current practices, in terms of validation and verification, are briefly presented and discussed (2.2). Lastly, the concepts of autonomic computing, more specifically, self-healing, are presented (2.3).

2.1 Node-RED

Node-RED is an open-source mashup-based[4] approach for developing IoT systems. Its "programs" are a set of *flows*, which consist of *nodes* connected by *wires*. Several *node templates* are usually provided that can be used (*e.g.*, drag-and-dropped) into a *flow canvas*. Once the developer creates or updates a flow, it must be *deployed*; a process that persists the new flow version and (re-)starts the whole system [7]. More recently, *flows* acquired the ability to be version controlled and exported. The portfolio of available *nodes* can be extended via *plugins* that implement new ones, either in (1) JavaScript, or (2) by the composition of existent *nodes* in the form of *sub-flows*. Input *nodes* typically subscribe to external services, listen for data on a specific port, or start processing HTTP requests. Once the data is processed by a given *node*, either from an external service or from an upstream *node*, a method is called with the resulting data on downstream *nodes* that can either generate additional events or push the results to outside services or systems [7]. Mashup tools are known to lower the barrier of application development significantly [24].

Despite its features and popularity, this tool still presents several limitations to our objective. There are no proper mechanisms for debugging and testing

[4] Mashup-based developed systems are the result of composing or mashing up existing services, components, and devices [26].

flows, beyond adding special nodes having logging capabilities. The message passing mechanism is not typed, which means simple connection errors are not detected before they are deployed. Meta-facilities, such as *reflection* and *reification*, are not available for usage in the flows, which might be due to it not leveraging the usage of a formally defined meta-model as a way of representing its abstractions [27]. The tool is also designed as a centralized *orchestrator*, in the sense that every flow — particularly every message passing activity — must be executed by it, even if several nodes gather or publish their information to external systems. One contributing factor to this limitation is the non-usage of model-based techniques, which leads to a platform-dependent specification, hindering the ability to generate target-specific code. Its design favors the modeling of the system's overall behavior as a *dataflow*, but the behavior of each particular component is mostly opaque and must be implemented manually. As such, it is harder to inspect, simulate, analyze, and change flows as a whole when compared to model-based systems — including during *runtime*.

The result is that, although Node-RED presents an easy platform to prototype simple systems, it quickly falls behind once the complexity starts increasing. Ray's survey findings [29] concur with our analysis, arguing that although several domains of applications already take great advantage of the use of recent advances in Visual Programming Languages, the emerging field of IoT is still lingers far behind other sectors.

2.2 Validation and Verification of IoT Systems

The complexity of our target systems affects not only their design and development processes but also implies a greater complexity of their verification and validation procedures. Traditional approaches for testing software-only systems are mostly limited and insufficient by overlooking fundamental factors about interaction with the real world, and mostly ignoring the hardware counterpart [9]. Of the available solutions, most focus solely on a specific platform, language, or standard, hindering overall improvement or extension, and do not provide out-of-the-box functionality [9]. The lack of IoT-specific testing systems can also lead to the adoption of poor testing practices; a closer examination allows the identification of recurring behaviors in these applications and a set of corresponding testing strategies [10]. Pontes et al. [25] proposed a pattern-based approach to IoT testing by identifying five specific test patterns, namely: *Test Periodic Readings*, *Test Triggered Readings*, *Test Alerts*, *Test Actions*, and *Test Actuators*. They claim that once these are available as test patterns, the overall process of testing becomes easier, as they can be reused to test recurrent behaviors in different scenarios.

2.3 Self-healing Systems

Ghosh et al. [14] describe systems with self-healing capabilities to be those that can deal with disruptions in their operation by (1) detecting system failures and possibly diagnosing the root cause of the problem, (2) determining a fix (*i.e.*, maintenance of health), and (3) recovering (even if only to a less capable but safe and healthy state). Self-healing may use models (*external* or *internal*) that monitor the system's behavior (*probes*), allowing it to adapt to environmental or operational circumstances. These approaches can be *intrusive*, if implemented internally within the system itself, or *non-intrusive*, if they consider the guarded system as a complete unit; they are *closed-loop* when they try to avoid all *a priori* known failure sources (*i.e.*, all possible states are known before recovery), or *open-loop* otherwise [28]. The typical recovery mechanisms employed include reconfiguration and replication of components (hardware and software) and degradation of the quality-of-service (QoS) [1].

3 Related Work

Athreya et al. [5] suggest devices should be able to manage themselves both in terms of configuration (self-configuration) and resource usage (self-optimization), proposing a measurement-based learning and adaptation framework that allows the system to adapt itself to changing system contexts and application demands. Although their work has some considerations about resilience to failures (*e.g.*, power outages, attacks), it does not address self-healing concerns.

The concept of *responsible objects*, introduced by Angarita et al. [3], states that *things* should be self-aware of their context (passage of time, the progress of execution and resource consumption), and apply *smart* self-healing decisions taking into account component transaction properties (backward and forward recovery). Their approach shows limitations, *viz.* (1) when applied to time-critical applications, as it is not clear how much time we should wait for a transaction to finish, (2) some processes, such as those triggered by emergencies, cannot be compensated, and (3) when is it acceptable to perform *checkpoints* in a continuously running system that cannot be *rolled-back*? It also disregards the typical capability of devices (*e.g.*, limited memory, power) that might challenge the implementation of transactions.

Aktas et al. [1] are amongst the first to purpose runtime verification mechanisms to identify issues by resorting to a complex event processing (CEP) technique and *"applying rule-based pattern detection on the events generated real-time"*. They do not address *self-healing* and only convey a summary of problems or possible problems to human operators. Leotta et al. [20] also present runtime verification as a testing approach by using UML state machine diagrams to specify the system's expected behavior. However, their solution depends on the definition of a formal specification of the complete system, which is unfeasible for highly-dynamic IoT environments (*e.g.*, dynamic network topology).

We could not find any work that focuses on bringing runtime verification mechanisms for visual programming environments. This is not unexpected, as Leotta et al. [20] point out that *"software testing (in IoT) has been mostly overlooked so far, both by research and industry,"* and later corroborated by Seeger et al. [30], claiming that most of the research being conducted in visual programming for IoT has been disregarding failure detection and recovery.

4 System Architecture and behavior

We encapsulate runtime verification and self-healing mechanisms by extending Node-RED with new *nodes*. The following subsections detail our approach for those that are subsequently used in the validation scenarios (Section 5.1), but they are part of a more extensive palette [11].

4.1 Visual Runtime Verification

Node-RED has several limitations regarding testing and debugging of *flows*, from not providing *out-of-the-box nodes* capable of doing these tasks, to some design decisions of the programming environment itself.

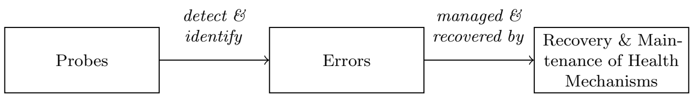


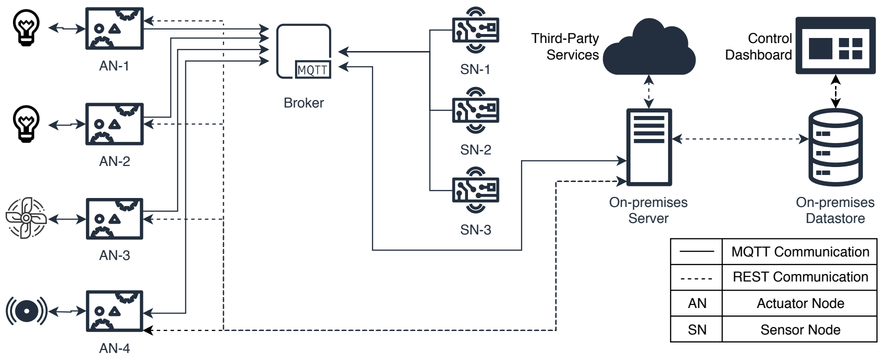
Fig. 1. Self-healing sequence. The use of runtime verification *probes* the system for errors, and the self-healing is accomplished by the activation of system recovery and/or maintenance of health mechanisms.

Regarding runtime verification capabilities, we created *nodes* that allow inspecting the system under test (SUT), *i.e.*, *probing* the system (Fig. 1), including the test patterns presented in Pontes et al. [25] and detailed in Section 5.1. Some devices and services (*e.g.*, message brokers, datastores, third-party services) can only be tested by implementing *black-box* reachability checking, such as the new `MQTTBrokerTimeout` node that asserts if the broker is still alive.

4.2 Visual Self-healing Approach

Following the self-healing loop described by Psair et al. [28], our *detection* component is composed by nodes that allow *runtime testing* and provide *diagnosis* information (Fig. 1), after which the *recovery* process is accomplished by nodes that implement *maintenance of health* and *recovery* mechanisms:

Replacement. Replace a faulty component with a duplicate spare one;
Balancing. Reduce or manage the load of a component to avoid damage;
Isolation. Isolate the failing component to keep the system in a *healthy state*;
Persistence. Assume that a failure does not cause further system degradation;
Redirection. Change the flow during a failure to a recovery routine and then back to the original;
Relocation. Move a system component (along with its dependencies) from a faulty host to a healthy one;
Diversity. Switch between different approaches or processes during runtime.



——	MQTT Communication
-----	REST Communication
AN	Actuator Node
SN	Sensor Node

Fig. 2. System component diagram, showing the main system parts, along with the different devices (actuators and sensors) and the enabling communication protocols.

Supporting these features requires *meta-facilities* that allow changing a system's behavior during runtime. As Node-RED does not formally provide them, we found a workaround by resorting to its *external* REST API from *inside* our nodes, thus gaining the ability to create, delete and change the configuration of *flows* and other *nodes*. This is exemplified by the `SetFlowStatus` node, which allows toggling flows *on* and *off*, thus providing the necessary capabilities for *redirection*, *replacement*, and *isolation*. We were then able to change between instances of message brokers and create a *balancing* mechanism. Other self-healing mechanisms were implemented by adding *secondary* flows and *sub-flows*, that are triggered when some precondition is met.

5 Experimental Scenarios and Results

Validating new solutions for runtime verification and self-healing requires scenarios representative of the characteristics, issues, and challenges of real-world IoT environments, such as heterogeneity and real-time needs. We carried experiments on *SmartLab*, an experimental *testbed* with four actuators and three sensing devices (each having more than one sensor) deployed in a laboratory (Fig. 2), responsible for a set of user-interaction features.

5.1 Scenarios

We devised three scenarios to demonstrate both the necessity of runtime verification as well as self-healing mechanisms. Although these scenarios do not cover all possibilities, we believe them to be sufficient to show the complexity, challenges, and, in this case, Node-RED limitations and trade-offs.

Unavailability of Message Broker. MQTT is the base of most of our *SmartLab* communications; thus, it needs a message broker. Typically, the defined flows are triggered when a new message is received (the flow subscribes to a specific

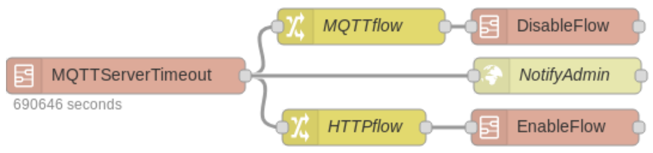


Fig. 3. `MQTTServerTimeout` example, for detecting and healing (using a replacement strategy) a potential unavailability of the broker.

topic). In this scenario (*cf.* Fig. 3), the message broker is both the *bottleneck* and a *single point of failure* (SPOF) of the system; if it fails, the functionality of the system is compromised. To verify its availability (*i.e.*, health status), a *heartbeat* pattern was followed: when the broker stops sending its periodic signal, it is assumed that some fault occurred. The same logic can be easily applied to other publish-subscribe protocols. When this kind of fault is detected, a *redirection strategy* is followed, ensuring the continuation of communications. In our scenario, we trigger a change from the *MQTT-dependant flow* to the alternative *HTTP-based flow*.

Erroneous Sensor Readings. *SmartLab* relies on the readings from different sensors so that it can act according to user-defined rules. As an example, if smoke is detected, an alarm or another notification mechanism should be triggered (and possibly trigger some contention mechanism like sprinklers). These procedures depend on the *timeliness* and *correctness* of readings. Sensor malfunctioning can display an array of different behaviors, such as outputting *out-of-bound* or *out-of-spec* values; these can lead to wrong decisions and may end up having nefarious effects to the point of impacting the well-being of humans. Several strategies can be used separately or in combination to detect sensor malfunction. Sensors that provide periodic readings can be verified by analyzing the expected periodicity (*cf.* Section 2.2). Other errors, such as *out-of-bounds* and *out-of-spec* readings require customized verification and tailored failure conditions. Fortunately, these are usually available; *e.g.*, the DHT11 temperature/humidity sensor is capable of readings ranging from 0°C to 50°C, and 20% to 80% humidity. Values outside these ranges should be considered erroneous by default. In this scenario, an *isolation strategy* is followed; when an *out-of-spec* problem is detected, the readings are ignored via the `TestAndFilter` node. In the presence of redundant sensors, other readings may still be used by the system; otherwise, all the actuating components that depend on that sensor cease their activity (Fig. 4).

Connectivity Issues. Devices that are part of our *SmartLab* provide HTTP *and* MQTT connectivity. These devices (especially actuators) depend on receiving messages to work as supposed. However, in some situations, the devices are not accessible by the protocol used by default (*e.g.*, MQTT) due to connectivity disruptions, protocol *bugs*, or other reasons, thus becoming inaccessible and eventually causing problematic side-effects (*e.g.*, sprinklers not turning on in the presence of a fire). In this case, a verification can be carried after a certain amount

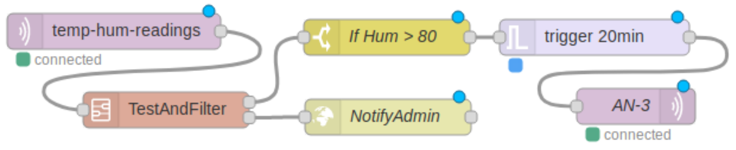


Fig. 4. `TestAndFilter` example in a flow that triggers an actuator if the humidity is above 80%, but verifies for correct sensor readings beforehand.

of time (*cf.* Section 2.2), asserting if the request has been processed by the device, preferentially using an alternative communication protocol. As an example, after a state change request message is sent to an actuator via MQTT, one could request its status, after a given time, to verify if the reported state corresponds to what was expected. Fixing scenarios in which the state of the system does not correspond to the expected, requires a *diversity strategy*. Having *things* that are capable of using different protocols allows us to adapt by dynamically switching to the most stable one given the systems' conditions (although usually incurring in a trade-off, such as the differences in energy consumption between MQTT and HTTP). As an example, if the light controlling device does not turn on the lights, as requested by the MQTT broker, a second request is made to the same device, this time using HTTP. This only can be implemented if both the device and the system can communicate using several different protocols. For this, we implemented a `TestAction` node that connects to the trigger and actuator *nodes* and checks if actions are triggered correctly. If not, a secondary flow is triggered, repeating the failed request using a different protocol. The resulting scenario implementation is depicted in Fig. 5.

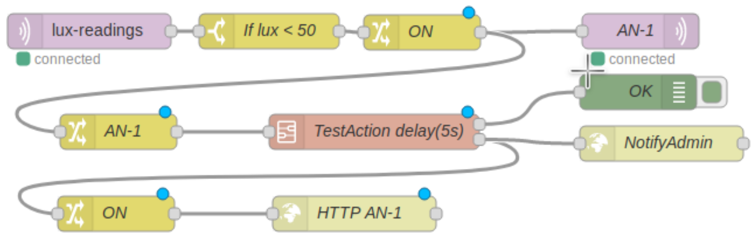


Fig. 5. `TestAction` example, where a verification is made to check if the lights turn on (request sent via MQTT) after a given interval, by checking if the luminosity lowers below 50 lux. If not, a secondary flow sends a new *on* request via HTTP.

5.2 Results

We showed improvements to *SmartLab* reliability and dependability both by detecting failures as they happen and recovering or maintaining the systems' health.

Node-RED does not provide any *out-of-the-box* solution for dealing with failing components, nor to dynamically change the system's behavior during runtime, which is essential to enable *self-healing*. After adding such functionalities via new nodes, users can now leverage these new capabilities. Our first example scenario shows how it becomes possible to test and recover from a SPOF (exemplified as a message broker failure). The same method could be used to deal with other SPOFs, including failures of Node-RED itself, with a `RedundancyManager` node that activates *duplicated* and *inactive* flows on a different Node-RED instance (provided one is available). The second scenario shows how to isolate a system's component to ensure that its misbehaviors do not compromise the system as a whole. The last scenario shows how we can now manage several (redundant) communication protocols as an enabler of *self-healing* mechanisms, and the importance of continuously asserting the actuators outcome.

6 Discussion

Ensuring the dependability of software systems has been the goal of most fault-tolerance research in the past years [6]. In IoT, ensuring systems are secure, reliable, and compliant is becoming a paramount concern due to the recent increase in safety-critical applications. Fault-tolerance becomes more challenging due to several factors, including, but not limited to: (1) the high heterogeneity of devices, (2) the interaction and limitations of systems deployed in a physical world, (3) the fragmentation of the field, ranging from the unusually high number of communication protocols, to the different and competing standards, and (4) the intrinsic dependability on hardware that might simply fail [2]. Moreover, in a perfect environment, every actuator should possess a monitoring sensor capable of verifying its intended end state; however, real-world cost efficiency might limit their availability to critical components.

The pervasiveness and complexity of IoT have contributed to the rise of visual programming, in particular Node-RED, as the go-to solution (see Section 1). Nevertheless, as it slowly permeates our lives, it becomes crucial to ensure proper functioning through self-verification and self-recovery features: *self-healing*. Although previous work attempted to tackle runtime verification and self-healing mechanisms to specific IoT systems (see Section 3), none was found to provide this kind of feature in a visual environment. Previous work also relies heavily on new systems (*e.g.*, rule-based monitoring services and CEP approaches), without attempting to integrate into the existent ecosystem of tools and platforms.

Although we chose to extend Node-RED due to its popularity, several challenges limit its potential concerning our use-cases (or introduce unnecessary accidental complexity). We already discussed some in Section 2.1, but while implementing our test scenarios (Section 5.1), the following issues became disproportionately prominent, namely:

Support for labels and annotations: Nodes do not visually provide sufficient information about their connectors and internal status, making flows harder to construct, debug, and adapt. Most (if not all) nodes configuration cannot be set or changed by other nodes. A solution similar to Fig. 6 seems more useful, not only in presenting this information but also in terms of flexibility regarding our goals;

Multiple inputs: Although Node-RED supports several outputs per node, they cannot have differentiated inputs (see Fig. 6). This poses both a cognitive and technical difficulty in defining and readjusting the behavior of nodes both during the design and runtime phases, including the configuration of test conditions and recovery measures;

Types and static analysis: Nodes do not have the notion of types; this allows the user to incorrectly connect two nodes, where the destination expects a different data type than the one sent by the origin one. This leads to common (and simple) errors that only make themselves noticeable after deployment of the whole flow, possibly introducing severe inconsistencies in the system;

Debugging: Besides the provided logging capabilities of Node-RED, using *debug nodes*, no other debugging technique is available. This means that breakpoints, *node* inspection, value history, and other apparatus are absent, severely hindering the ability for the developer to understand what went wrong in the internal logic of a node;

Meta-programming: Formal mechanisms of *introspection* and *reification*, essential for effective meta-programming, are non-existent. This limits the possibility of adjusting *flows* in runtime and forces us to rely on external APIs that were not designed for this particular purpose and which might easily break.

Despite these limitations, it was possible (up to a certain extent) to fulfill our goals mostly by using its visual notation, as seen in Section 5 and discussed in Section 5.2. It should be noted that all implemented strategies fall into the *forward error recovery* category, *i.e., "continue from an erroneous state by making selective corrections to the system state"* [17]. Exploration of *backward error recovery* techniques is harder due to the dependency of system state *checkpoints*, that needs to capture a mix of device internal states, concurrent communication protocols messages, and controller state.

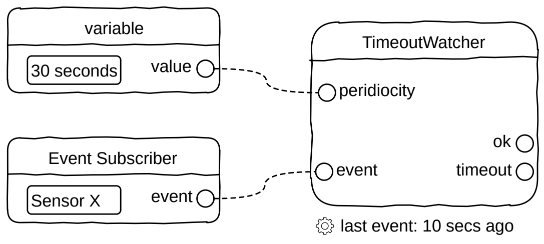


Fig. 6. Mockup of possible *node* interface with annotations, labels and multiple inputs/outputs.

To further improve the self-healing capabilities of systems such as the presented *SmartLab*, devices should have extra features such as diverse communication channels (*e.g.*, Wi-Fi and ZigBee), remote management capabilities (*e.g.*, independent watchdogs that allow to gracefully restore a device), and capability announcement, which would empower dynamic usage of redundant devices. We observe these features are mostly absent from consumer-grade devices, most probably due to cost efficiency.

Several challenges remain unaddressed by this work, such as (1) dealing with concurrent inputs that can lead to unexpected states (*e.g.*, the system decides to turn on the lights and the user manually turns them off), which may result in false assertions by the runtime verification mechanisms, (2) auto-discovery and configuration of new devices in the system (*e.g.*, a new mobile device can be used as a redundant sensing node while it remains in the system network), and (3) what are the reasonable operational states that the system should converge to in the case of failure (*e.g.* if the system has to decide between shutting down the smoke alarm or the surveillance system, which one should take prevalence?). Supporting and articulating with other *self-** aspects is also an open challenge towards fully autonomic systems; this includes *self-protection*, *self-optimization*, and *self-configuration* [13].

7 Conclusion

IoT systems are perhaps one of the most significant examples of heterogeneous architectures in existence. Different protocols, different application stacks, different integration services, and different orchestration engines, all must come together in a technological solution that allows both an organic growth from end-users, as well as dealing with security and privacy concerns at unprecedented levels. The consequence is that the system is required to keep functioning at minimal levels, even when parts of it become non-compliant, faulty, or even under attack. Requiring the end-user to address these challenges is unrealistic, as most of them are not developers. Even most system integrators cannot keep up with the pace of release devices, which very seldom adhere to open standards.

In this paper, we argue that an IoT system that attempts to tackle the presented challenges must be capable of *self-healing*. This is not a small feat, as most of the research being conducted in integration tools for IoT recurrently disregard failure detection and recovery. We fulfill these desiderata with SHEN, Self-Healing Extensions for Node-RED. As this very popular tool lacks built-in testing and self-healing capabilities, we use it as a case-study for common failure and recovery scenarios, and (1) show how to leverage *meta-programming* techniques to allow self-modification of *flows* via a custom *plugin*, (2) explore common self-healing patterns and how they can be solved by such techniques, (3) provide them as reusable *nodes* for others to incorporate in their systems, and (4) discuss which challenges remain open and which might need rethinking architectural and design decisions.

To validate our claims, we applied SHEN to the existing *SmartLab*, and proceed to show its behavior for three different scenarios, *viz.* (1) Unavailability of Message Broker, (2) Erroneous Sensor Readings and (3) Connectivity Issues. We conclude that we can improve the system's reliability and dependability, both by being able to detect failing conditions, as well as reacting to them by self-modification of defined *flows*. Future work includes (a) the extension of the SHEN palette with more runtime verification's and self-healing mechanisms, and (b) case studies over various degrees of systems complexity, and in different contexts and scales.

Acknowledgement. This work was partially funded by the Portuguese Foundation for Science and Technology (FCT), under the research grants SFRH/BD/144612/2019 and SFRH/BD/115358/2016.

References

1. Aktas, M.S., Astekin, M.: Provenance aware run-time verification of things for self-healing Internet of Things applications. Concurrency Computation **31**(3), 1–9 (2019)
2. Aly, M., Khomh, F., Gueheneuc, Y.G., Washizaki, H., Yacout, S.: Is fragmentation a threat to the success of the internet of things? IEEE Internet of Things Journal **6**(1), 472–487 (Feb 2019)
3. Angarita, R.: Responsible objects: Towards self-healing internet of things applications. Proceedings - IEEE International Conference on Autonomic Computing, ICAC 2015 pp. 307–312 (2015)
4. Ashraf, Q.M., Habaebi, M.H.: Introducing autonomy in internet of things. In: 14th International Conference on Applied Computer and Applied Computational Science (ACACOS'15) (2015)
5. Athreya, A.P., DeBruhl, B., Tague, P.: Designing for self-configuration and self-adaptation in the Internet of Things. Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, COLLABORATECOM 2013 pp. 585–592 (2013)
6. Avizienis, A., Laprie, J.C., Randell, B.: Fundamental Concepts of Dependability. Technical Report Seriesuniversity of Newcastle Upon Tyne Computing Science **1145**(010028), 7–12 (2001)
7. Blackstock, M., Lea, R.: Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED). In: Proceedings of the 5th International Workshop on Web of Things - WoT '14. pp. 34–39 (2014)
8. Delicato, F.C., Pires, P.F., Batista, T., Cavalcante, E., Costa, B., Barros, T.: Towards an iot ecosystem. In: Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems. pp. 25–28. SESoS '13, ACM (2013)
9. Dias, J.P., Couto, F., Paiva, A.C.R., Ferreira, H.S.: A brief overview of existing tools for testing the internet-of-things. In: IEEE International Conference on Software Testing, Verification and Validation Workshops. pp. 104–109 (April 2018)
10. Dias, J.a.P., Ferreira, H.S., Sousa, T.B.: Testing and deployment patterns for the internet-of-things. In: Proceedings of the 24th European Conference on Pattern Languages of Programs. EuroPLop '19, ACM (2019)

11. Dias, J.P.: jpdias/node-red-contrib-self-healing: Replication package for ICCS 2020. (Apr 2020). https://doi.org/10.5281/zenodo.3746414
12. Dundar, B., Astekin, M., Aktas, M.S.: A big data processing framework for self-healing internet of things applications. In: 2016 12th International Conference on Semantics, Knowledge and Grids (SKG). pp. 62–68. IEEE (2016)
13. Ganek, A.G., Corbi, T.A.: The dawning of the autonomic computing era. IBM systems Journal **42**(1), 5–18 (2003)
14. Ghosh, D., Sharman, R., Rao, H.R., Upadhyaya, S.: Self-healing systems — survey and synthesis. Decision Support Systems **42**(4), 2164 – 2185 (2007), decision Support Systems in Emerging Economies
15. İnçki, K., Arı, İ., Sözer, H.: Runtime verification of iot systems using complex event processing. In: 2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC). pp. 625–630. IEEE (2017)
16. Janssen, P., Erhan, H., Chen, K.W.: Visual dataflow modelling - some thoughts on complexity. In: Proceedings of the 32nd eCAADe Conference (2014)
17. Jia, W., Zhou, W.: Reliability and replication techniques. Distributed Network Systems: From Concepts to Implementations pp. 213–254 (2005)
18. Kopetz, H.: Real-Time Systems. Real-Time Systems Series, Springer (2011)
19. Krupitzer, C., Roth, F.M., VanSyckel, S., Schiele, G., Becker, C.: A survey on engineering approaches for self-adaptive systems. Pervasive and Mobile Computing **17**, 184–206 (2015)
20. Leotta, M., Ancona, D., Franceschini, L., Olianas, D., Ribaudo, M., Ricca, F.: Towards a Runtime Verification Approach for Internet of Things Systems. In: Proceedings of the International Conference on Web Engineering, vol. 11153, pp. 83–96. Springer International Publishing (2018)
21. Leucker, M., Schallhart, C.: A brief account of runtime verification. Journal of Logic and Algebraic Programming **78**(5), 293–303 (2009)
22. Minerva, R., Biru, A., Rotondi, D.: Towards a definition of the internet of things (iot). IEEE Internet Initiative **1**, 1–86 (2015)
23. Morin, B., Harrand, N., Fleurey, F.: Model-Based Software Engineering to Tame the IoT Jungle. IEEE Software **34**(1), 30–36 (2017)
24. Patel, P., Cassou, D.: Enabling high-level application development for the internet of things. Journal of Systems and Software **103**(C), 62–84 (2015)
25. Pontes, P.M., Lima, B., Faria, J.a.P.: Test patterns for iot. In: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation. pp. 63–66. A-TEST 2018, ACM (2018)
26. Prehofer, C., Chiarabini, L.: From IoT Mashups to Model-based IoT. W3C Workshop on the Web of Things (2013)
27. Prehofer, C., Chiarabini, L.: From internet of things mashups to model-based development. In: 2015 IEEE 39th Annual Computer Software and Applications Conference. vol. 3, pp. 499–504. IEEE (2015)
28. Psaier, H., Dustdar, S.: A survey on self-healing systems: Approaches and systems. Computing (Vienna/New York) **91**(1), 43–73 (2011)
29. Ray, P.P.: A Survey on Visual Programming Languages in Internet of Things. Scientific Programming **2017**, 1–6 (2017)
30. Seeger, J., Bröring, A., Carle, G.: Optimally self-healing iot choreographies (2019)
31. Vermesan, O., Friess, P., Guillemin, P., Gusmeroli, S., Sundmaeker, H., Bassi, A., Jubert, I.S., Mazura, M., Harrison, M., Eisenhauer, M., et al.: Internet of things strategic research roadmap. Internet of things-global technological and societal trends **1**(2011), 9–52 (2011)