Complex Patterns of Failure: Fault Tolerance via Complex Event Processing for IoT Systems

Alexander Power and Gerald Kotonya
School of Computing and Communications
Lancaster University
Lancaster, United Kingdom
{a.power3, g.kotonya}@lancaster.ac.uk

***Abstract*—Fault-tolerance (FT) support is a key challenge for ensuring dependable Internet of Things (IoT) systems. Many existing FT-support mechanisms for IoT are static, tightly coupled, and inflexible, and so they struggle to provide effective support for dynamic IoT environments. This paper proposes Complex Patterns of Failure (CPoF), an approach to providing FT support for IoT systems using Complex Event Processing (CEP) that promotes modularity and reusability in FT-support design. System defects are defined using our Vulnerabilities, Faults, and Failures (VFF) framework, and error-detection strategies are defined as nondeterministic finite automata (NFA) implemented via CEP systems. We evaluated CPoF on an automated agriculture system and demonstrated its effectiveness against three types of error-detection checks: reasonableness, timing, and reversal. Using CPoF, we identified unreasonable environmental conditions and performance degradation via sensor data analysis.**

***Index Terms*—internet of things, fault tolerance, dependability, complex event processing, automata**

I. Introduction

The *Internet of Things* (IoT) is the latest evolution of the Internet that aims to facilitate *thing-to-thing* interactions by embedding virtual and physical objects with electronics, sensors and connectivity, enabling them to achieve greater value and services [1], [2]. An important challenge to realize IoT is how to provide a dependable infrastructure for the billions of expected devices and deliver their intended services without failing in unexpected and catastrophic ways [3].

Dependability is threatened by faults and errors that contribute to the occurrence of service failures, where a system can no longer provide its service as intended [4]. This problem can be addressed by designing IoT systems that support *fault tolerance* (FT) to prevent service failures [5]. Lee et al. [6] define four phases of FT, namely:

1) **Error Detection**. A fault activation manifests as one-or-more system errors, and these must be detected.
2) **Damage Assessment and Confinement**. The damage caused by a fault must be assessed in order to identify and isolate all errors caused by it.
3) **Error Recovery**. The system must execute error-recovery strategies to move the system into an error-free state wherein normal system operations can continue.
4) **Fault Treatment and Continued Service**. Techniques may still be required to ensure that faults do not recur in a system despite successful error recovery.

Current implementations of FT support in IoT are static, tightly coupled, and inflexible. For example: (1) they are designed for a specific architecture and application [7], [8]; (2) they do not scale beyond small (decentralized) solutions [9], [10]; and (3) they provide solutions to specific faults, such as link failures [11]. This is problematic because IoT systems are expected to continuously evolve in order to handle new services, features, and devices that had not been anticipated when the system was first designed, and FT support needs to evolve with it. Additionally, many adopt FT solutions that are already widely explored in *distributed systems* (DS), such as hardware redundancy [7], check-pointing [12], and traffic re-routing [11]. Their error-detection approaches are implemented in an application-specific manner, wherein faults are identified *a priori* by system designers, with checks that only detect errors in highly specific scenarios and contexts.

IoT must go beyond DS solutions and further consider the importance of *context awareness* in FT. IoT systems are highly associated their physical environment, and so FT support should be able to detect erroneous phenomena occurring within the context of the environment, and also use contextual information for error recovery. For example, context awareness has been proposed to optimize waste management by installing level sensors on waste bins so that waste trucks can build optimized routes to reduce fuel consumption [13].

Complex Event Processing (CEP) is used in research and industry to identifying complex, high-level situations (*composite events*) by defining rule-based patterns in stream data (*primitive events*). It is considered the paradigm of choice for monitoring and reactive applications [14], [15], making it the ideal platform to provide the FT-support features outlined above. The focus on context awareness and adaptability in IoT calls for novel FT-support approaches that can handle a new range of situational faults that go beyond those found in DS. To the best of our knowledge, CEP has not been considered as a means of providing FT support in IoT systems before.

Our contribution is twofold. Firstly, we define the *Vulnerabilities, Faults, and Failures* (VFF) framework that generically describes a system defect and its potential effect on a system. Secondly, we propose *Complex Patterns of Failure* (CPoF), an approach to FT support that uses CEP as the means of detecting and recovering from system errors, where error-detection events are defined as *nondeterministic finite automata* (NFA).

Together, VFF systematically categorizes defects so that, for each one, there are corresponding NFA(s) to handle it in CEP. In this paper, we use NFA to implement three types of error checking: (1) whether data is *reasonable*, given some criteria; (2) whether data is *timely*, given them time threshold; and (3) performing *reversal* checks, to consider whether there is a correlation between two separate error events. We prescribe that error events be recursively fed back into the CEP data stream for reuse in defining more complex error-detection events, which we call *complexity via recursion* (CvR).

This papers sets out to discover whether we can: (1) implement error detection via CEP; (2) provide effective error recovery via CEP; and (3) define a taxonomy of generic NFAs to implement application-agnostic FT support, applicable to all IoT systems. The rest of the paper is as follows. Section II discusses related work. Section III covers VFF. Section IV explores NFAs for error detection and recovery. Section V discusses CvR. Section VI evaluates CPoF with an IoT agricultural system. Section VII concludes our work.

II. Related Work

A core challenge when analyzing stream data from things in the physical world is how to infer the occurrence of interesting and *complex* situations in the environment. We focus on CEP systems that are based on NFA because it is the established mechanism upon which most CEP systems are based [16]. In literature, the language model of existing CEP systems share a variety of common operators, namely [17], [18]:

- **Logic Operators**. These define rules by combining several items (e.g. via conjunction, disjunction, negation).
- **Sequences**. These are similar to logic operators but items are order dependent i.e. they are satisfied when detected in a specified order. **Iterations** are a special case, where the sequence length is not *a priori* known, enabling unbounded sequences.
- **Windows**. These involve limiting portions of input flow to those only within a given timeframe, which also ensures the termination of unbounded iterations.
- **Event Selection**. Events can be dispersed over many input streams and, thus, are not always contiguous, especially in IoT environments. We focus on the **skip till next match** (STNM) selection policy, where irrelevant events are skipped until the next relevant event occurs.

A key reason for using CEP to provide FT support in IoT is because of its ability to realize *context-aware* computing. A *context* is any information that can characterize the situation of an entity (i.e. people, places, objects), and a system is context-aware if it uses context to provide relevant information and services to the user [19]. The system stores context information linked to sensor data so that data interpretation and *machine-to-machine* communication can be done easily [1]. Hasan et al. [20] considered context awareness over large-scale sensor networks via dynamic enrichment of information flows, which are combined with CEP, as the means to realize situation awareness. Barbero et al. [21] proposed the *Concept Reply* IoT platform to provide support for context-aware application deployment throughout the low-level and middleware layers of IoT systems. It contained a reasoning framework and event-based processing agents that incorporated CEP for content-based filtering. Nallaperuma et al. [22] proposed *Incremental Knowledge Acquisition and Self Learning* (IKASL), an unsupervised incremental learning algorithm for detection and adaption of concept drifts in data by monitoring changes in context i.e. location, time, activity and identity. They evaluated its efficacy on a motor-traffic dataset to detect drifts in the number of vehicles on the road.

Maarala et al. [23] focused on the issue of providing and acquiring knowledge in IoT environments with a study on Semantic Web technologies that can facilitate context-awareness, interoperability, and reasoning in IoT. They identified that the publish/subscribe message-exchange scheme supports topic and content-based message routing and aggregation methods, to enable context-based information fusion from multiple heterogeneous data sources for reasoning and integrating knowledge from diverse application domains. This supports our decision to use CEP systems for context-aware computing because they extend the functionality of publish/subscribe systems by increasing the expressive power of the subscription language to consider complex event patterns that involve the occurrence of multiple, related events [17].

FT has been widely explored in IoT for many diverse applications. Liu et al. [24] proposed a framework to monitor IoT systems in an oil field to handle outliers, stuck-at faults, and spikes in sensor data. Their approach used *statistics sliding windows* that created a series of windows, where the latest window contained recent sensor data that would regress into historical windows as new data arrived. Their evaluation was a simulation using real data from an oil field, which contained 751.68 million samples from 5800 sensors, and results showed that their system could detect 95% of the three data errors.

Choubey et al. [25] presented a smart home architecture where sensors were first analyzed for correlations so that, if some sensor data could be predicted by others, a neural network was trained to predict the data if a *crash* occurred. Hu et al. [12] presented a framework that enabled developers to implement FT support via user-defined exception handling (i.e. try-catch statements) in languages such as Java and Python. A snapshot of the sensor data was stored for check-pointing and a relevant error handler was notified when an error was detected. Gia et al. [7] explored FT in a healthcare scenario with a centralized architecture that consisted of low-level nodes, a gateway of *redundant* data sinks, and a back-end server that consumed data from sinks. Upon a data blackout, it pinged the node via an alternative sink, to ascertain whether the node or the sink had failed.

Javed et al. [26] proposed CEFIoT to provide FT using: (1) Docker, for consistency across cloud and edge; (2) Kafka, to replicate and buffer data when disconnected from the cloud; and (3) Kubernetes, for reconfiguration to handle hardware and network failures. Their evaluation used 5 Raspberry Pis with attached cameras that replicated images across devices. They tolerated two node failures as Kubernetes shifted processing

TABLE I
APPLICABILITY BETWEEN VULNERABILITIES, FAULTS, AND FAILURES FROM SECTION III (BULLET-POINTED). DEV REPRESENTS DEVELOPMENT FAULTS, PHY FOR PHYSICAL, AND INT FOR INTERACTION.

Vulnerability	Fault	Failure Semantics			
		Omission	Crash	Timing	Response
Hardware	Dev	•	•	•	•
	Phy		•		•
	Int		•		•
Software	Dev	•	•	•	•
	Phy				
	Int		•		•
Networks	Dev	•	•	•	•
	Phy	•	•	•	•
	Int	•	•	•	•
Payload	Dev		•		•
	Phy				
	Int		•		•
Environment	Dev				
	Phy				
	Int	•	•	•	•
Power	Dev				
	Phy		•		
	Int				
Human	Dev				
	Phy		•		•
	Int		•		•
Policy	Dev	•	•	•	•
	Phy		•		•
	Int		•		•

to different nodes. Kafka handled node damage by retrieving images from replica nodes.

III. CATEGORIZING VULNERABILITIES, FAULTS, AND FAILURES

The VFF framework is designed to consider the relationship between system *vulnerabilities*, *faults*, and *failures*. We identify how these three characteristics help to categorize defects so that, for each one, there are corresponding NFA(s) to handle it in CEP. In doing so, we can design modular, reusable error detection and recovery techniques to generically handle common system defects in all IoT systems. Each defect can be described as: *a vulnerability v, exploited by fault f, that may lead to failure s*. For the rest of this section, we discuss the framework's three attributes and summarize the applicability between them in Table I.

A. Vulnerabilities

The *eight-ingredient* (8I) framework was developed for conducting vulnerability analysis on internal and external aspects of a system and identifies that the reliability and security of communications is vital for continuous system operation [27]. It identifies eight 'ingredients' that represent different vulnerabilities that can manifest, namely:

- **Hardware**. Electronic and physical components that compose the network nodes (e.g. circuits, fiber optics, semiconductor chips).
- **Software**. All aspects of creating, maintaining, and protecting that code (e.g. physical storage, code development, testing, and delivery).
- **Networks**. Topological configurations of nodes, synchronization, redundancy, and physical and logical diversity.
- **Payload**. Information transported across the infrastructure (e.g. information interception, corruption).
- **Environment**. Physical spaces within which systems operate (e.g. harsh (weather) conditions, cell towers).
- **Power**. Power required for communications networks (e.g. internal power infrastructure, batteries).
- **Human**. System operators are, themselves, vulnerabilities (e.g. causing (un)intentional behaviors, physical and mental limitations, education and training).
- **Policy**. Agreements, standards, policies, and regulations defining the behavior between entities and governments.

For the remainder of the paper, we will refer to these ingredients as *vulnerabilities*. The presence of vulnerabilities in IoT systems can lead to the existence of faults which, in turn, can cause service failures (Section III-C).

B. Faults

Isermann [28] defines a *fault* as an unpermitted deviation of at least one system property from the acceptable, usual, standard condition. A fault is *active* when it produces one-or-more errors; otherwise it is *dormant*. An activation is the application of an input to a component that causes a dormant fault to become active [29]. Faults can be placed into three major groupings, namely: (1) **development faults**, ones that occur during system development; (2) **physical faults**, ones that affect hardware; and (3) **interaction faults**, ones introduced into the system from external sources [29].

C. Failures

A *service failure* is the permanent interruption of a system's ability to perform its required function under specified operating conditions [28]. A wide variety of reactive FT techniques have been explored in the IoT domain which fall within the five categories, namely: redundancy, migration, failure semantics, failure masking, and recovery [30]. *Failure semantics* categorize allowable server behaviors that occur in DS so that developers can understand the likely failures a system might exhibit, in order to develop relevant recovery strategies. They are as follows [31], [32]:

- **Omission**. When a server fails to respond to incoming messages, caused by failure to *send* or *receive* messages.
- **Crash**. When a server halts and no further service is provided by it. It is essentially an omission failure whereby *all* incoming messages receive no response.
- **Timing**. When a server's response is outside of the specified time interval, classified as *early* or *late*. If a response never arrives, then it is an omission failure.
- **Response**. When a server's response is incorrect. This manifests in the form of an incorrect *value* returned by the server, or as an incorrect *state transition*.

IV. ERROR DETECTION

An *error* is a deviation of a program operation from its exact requirements due to the presence of bugs that only appear

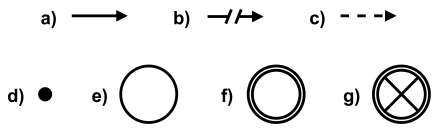

Fig. 1. NFA diagram symbols: (a) transition between states using STNM; (b) same as (a), omitting some intermediary states; (c) arrow pointing to composite event(s) produced after NFA acceptance; (d) a starting point; (e) a state; (f) an accepting final state; (g) a non-accepting final state.

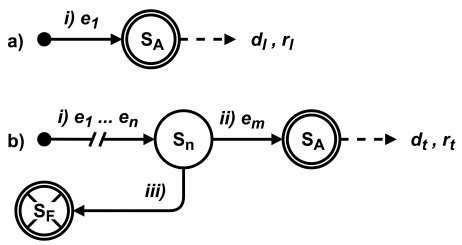

Fig. 2. Reasonableness NFAs: (a) limit checking; and (b) trend checking.

when a program is running or being tested [33]. An error is said to be *detected* if its presence is indicated by an error message/signal, whereas an undetected error is called *latent* [29]. As with faults and failures, we want to consider the generic categories into which all error-detection methods can be placed, so that faults can be generically paired with effective error-detection strategies. Bauer [34] prescribes eight product-attributable error categories, namely:

- **Field-Replaceable Unit (FRU) Hardware**. Part of electronic equipment that can be replaced without needing to repair the entire hardware (e.g. processor, disk failure).
- **Programming Errors**. Failures related to software design and development (e.g. memory leak, infinite loop, logic errors, crashes).
- **Data Inconsistency and Errors**. Massive amounts of complex data can lead to inconsistencies and errors (e.g. checksum errors, file corruption).
- **Redundancy Errors**. Failures related to the redundancy mechanisms in place to provide system robustness (e.g. failover to a failed FRU).
- **System Power**. Power disruption and its effects on the system (e.g. under/overvoltage, battery exhaustion).
- **Network Errors**. The degradation and failure of network communications facilities and infrastructure (e.g. dropped/corrupted IP packets, network outages, inconsistent real-time clocks).
- **Application Protocol Errors**. Differences between systems' implementations of a protocol (e.g. illegal message sequences, protocol version mismatch).
- **Procedures**. Performance by humans on the system (e.g. inadequate failure detection, diagnosis, recovery).

A. Automata Model

Our framework considers how error-detection events can be designed using NFA. Our automata model is similar to that described in [35] and we represent our NFA diagrams using the symbols in Figure 1. We define S as the set of all states in an automaton. For each state transition (Figure 1a,b), there is an event e_i that causes a transition to some state S_i, starting at state $S_1 \in S$. An event has a value e^v, representing its data; an origin e^o, representing where it was generated; and a timestamp e^t, representing when it was generated. We assume the STNM selection policy (Section II) because strict contiguity between events is ineffective due to IoT systems having high-volume, heterogeneous data.

The final state $S_A \in S$ is the *accepting* state that causes the CEP system to produce composite events. A dashed arrow (Figure 1c) points from S_A to the composite error-detection event d and an error-recovery event r. Event r is optional because recovery is not mandatory for each detected error. When S_A is reached, the set of all accepted events E is called an event's *pattern*.

The final state $S_F \in S$ is the *non-accepting* state that a NFA transitions into in order to halt (Figure 1g). S_F differs from S_A in that it does not produce any composite events. NFAs that have iterations (Section II) require *state clearance* to prevent an endless consumption of events (e.g. leading to out-of-memory errors). We consider two techniques to implement state clearance: (1) a *time window*, to transition to S_F after a time elapse occurs; and (2) an *until* predicate, to transition to S_F when fulfilled by an event.

B. Error-Detection Checks

Lee et al. [6] define seven error-detection checks: replication, timing, reversal, coding, reasonableness, structural, and diagnostic checks. We will use these checks as the basis for our low-level, 'simple' NFAs. Due to space constraints, we will explore three of these, discussed next.

1) Reasonableness: The reasonableness of events concerns whether they are acceptable based upon criteria envisaged by the system designer and implemented via the internal design and construction of the system [6]. We consider three types of data unreasonableness explored by Liu et al. [24], namely: (1) *outliers*, where e^v exceeds some threshold ϵ; (2) *stuck-at faults*, where the last n event values, V, are all equal; and (3) *spikes*, where some values in the last n events are drastically higher or lower than others, resulting in high variance.

a) Limit Checking: Detecting outliers is performed via limit checking, to check if e^v is 'within its limits'. In a NFA, this would simply require a predicate that checks if e_1 is *not* within some defined limits (Figure 2a), for example:

$$P(e) : \neg(\epsilon_{min} \leq e^v \leq \epsilon_{max})$$

If true, the NFA transitions to S_A and an error-detection event d_l is produced, optionally followed by error-recovery event r_l. The pattern for both d_l and r_l is $\{e_1\}$.

b) Trend Checking: Isermann [28] proposed calculating trend checking by taking the first derivative of the event value $f'(e^v)$, and then limit check as before. We apply this as:

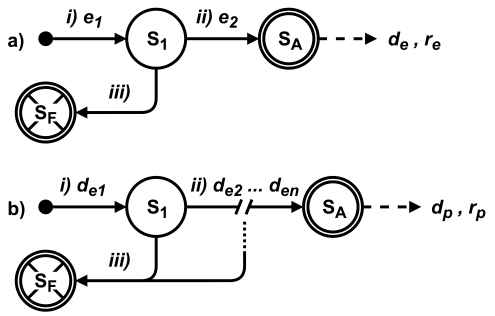

Fig. 3. Timing NFAs: (a) performance checking; and (b) persistence checking.

$$P(e) : \neg(\epsilon_{min} \leq f'(e^v) \leq \epsilon_{max})$$

If true, a trend has *not* been smooth, and thus can be considered unreasonable. We propose the NFA in Figure 2b for trend checking, and can be used for detecting stuck-at faults and spikes. To detect a trend, our proposed NFA calculates the slope between all relevant events that occur within time window t. If the slope between two events e_1, e_m, or an aggregate of n prior events $f(e_1, ..., e_n), e_m$, surpasses a slope threshold, error events d_t, r_t are generated. Otherwise, e_m is discarded and the NFA reattempts with some future e_m event, or halts on a time window elapse. This design enables spike detection by checking for exceptionally large trend changes between events. Stuck-at fault detection occurs if the trend is persistently 0.

2) Timing: A timing check is a simple implementation that detects when an operation fails to satisfy a specified time bound, and typically uses absolute or interval timers to invoke the detection mechanism [36]. These checks can address three scenarios: (1) where there exist two events e, e' and an 'unacceptable' time interval ϵ between them; (2) where one event e exists and an unacceptable time elapse ϵ that occurs *without* the next event e'; and (3) where a set of events E occur within some time bound ϵ. We explore these next.

a) Performance Checking: For this check, we want to identify the timeliness of events such that, if an event does not occur within time threshold ϵ since the last time, then an error-detection event d_e is produced, representing a *performance failure*, which is another term for a late timing failure (Section III-C). For example, we might want to identify the timeliness of events coming from a given origin (e.g. a sensor that is sending data at a constant rate). Event e_1 is first accepted, and a transition to S_A then occurs if the following predicate returns true (Figure 3a):

$$P(E) : e_2^o = e_1^o \wedge (e_2^t - e_1^t) > \epsilon$$

If $\leq \epsilon$, the NFA halts (Figure 3a-iii). To detect when a second event does not arrive at all, the CEP system still needs an e_2 event to reach S_A. A limitation with NFAs is that a *negation* (Section II) cannot be the final state transition. That is, an NFA cannot reach an accepting state by waiting for something to *not* happen. In literature, pruning NFAs can be accomplished using a periodically generated *null event*, $e^{\varnothing}$, that helps when reasoning about intervals between events [37]. Thus the time between event e_1 and $e_2 = e^{\varnothing}$ can be calculated instead.

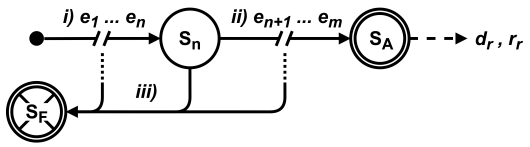

Fig. 4. Reversal NFA for correlation checking.

b) Persistence Checking: One of the most problematic issues to identify in DS is whether a device has crashed or is simply being untimely with its data [31]. In this scenario, there is a threshold of time t when a timing error is thought to be caused by data omission, and a later time $(t+h)$ when the error escalates to a more severe assumption of a crash failure, caused by the *persistence* of the symptoms of the underlying fault(s).

Persistence can be defined by three discrete categories, namely [29], [33]: (1) *transient*: arbitrary faults, bounded in time, that cause erroneous behavior for a short time before going away; (2) *intermittent*: faults that do not go away entirely, but instead oscillate between being active and dormant; and (3) *permanent*: faults that are assumed to be continuous in time. If the pattern of an error-detection event d does not make any reference to another, then we consider d to be *transient*, because it is independent from (i.e. it exists without knowledge of) other detected errors in the system. By always starting with the prior assumption of transience, different recovery strategies can be applied to handle the same underlying fault, if persistence is observed over time.

In Figure 3b, we consider how d_e from Figure 3a can be checked for persistence. For both intermittent and permanent persistence, the NFA accepts $n > 1$ events of type d_e to reach final state S_A. Halting occurs on a time window elapse. We propose that intermittent persistence be implemented as having $n > 1$ occurrences of d_e in time t, and permanent persistence as having $n' \geq n$ occurrences in time $t' \geq t$. The intuition behind this is that permanent persistence would have more occurrences over a longer period of time than an intermittent fault. Additionally, NFAs checking for permanent persistence might include an until predicate that halts the NFA if error recovery is ever successful. This is because a successful error recovery suggests that a fault is not permanent (e.g. successfully pinging crashed hardware is impossible).

3) Reversal: A reversal check is one that takes the output from a system and calculates what the input(s) should have been in order to produce that output, where the calculated inputs are used to compare with the actual inputs to check for an error [6]. This check has predominantly deterministic applications (e.g. reading back what was just written to disk). However, we consider how this check can be used in scenarios to check whether there is a relationship between two (sets of)

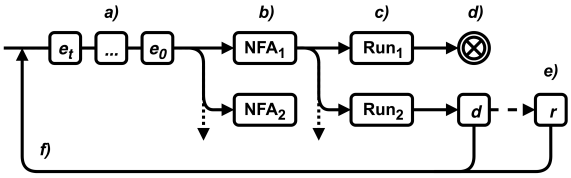

Fig. 5. The process of CvR in a CEP system.

events. That is, some system action(s) and/or event(s) lead to some other action(s) and/or event(s) as a consequence.

a) Correlation Checking: From an FT perspective, we want to identify whether, given $n \geq 1$ system events $e_1, ..., e_n$, there were $n \geq 1$ erroneous events $e_{n+1}, ..., e_m$ that occurred afterwards within a given time frame, or halt otherwise (Figure 4). If they do occur, we can infer that the erroneous event(s) were *caused by* the preceding event(s). This check helps to handle scenarios where an error *propagates* through a system and cause other errors [29].

V. Complexity via Recursion

Ascertaining the effectiveness of an error-recovery strategy requires post-recovery assessment and fault treatment to ensure that a fault does not persist. We propose that, instead of defining complex, monolithic NFAs to handle specific and well-defined error scenarios, we instead want to define simple NFA to handle low-level, 'atomic' errors, and have the error-detection and recovery events that are produced be *recursively* fed back into the CEP system and used in other more complex and application-specific NFAs. We call this CvR because it relies on 'event recursion' to define composite events of increasing complexity. The process of CvR is illustrated in Figure 5, as follows:

(a) A stream of events from e_0 to e_t enter the CEP system continuously over time.
(b) Each event is passed to the NFAs. When an event fulfills the predicate for the first state of the NFA, a copy of the NFA, called a *run*, is created.
(c) An event is passed to each run currently not in a final state or halted. This event might cause a state transition.
(d) A run might transition to a final state, producing an error-detection event d, or halt.
(e) Event d is passed to an error-recovery handler that will attempt to recover from d. It will produce an error-recovery event r detailing the actions taken to handle the error and whether they were successful or not.
(f) Events d, r are fed back into the CEP data stream to potentially be used in other NFAs.

The benefit of CvR is that it enables error events to be *modular* and *reusable*, freeing individual events to be used by an arbitrary amount of other NFAs, rather than being tightly coupled and monolithic. In particular, it enables: (1) error propagation detection by correlating error-detection events; and (2) system designers to define alternative recovery strategies in the event of unsuccessful recovery action(s).

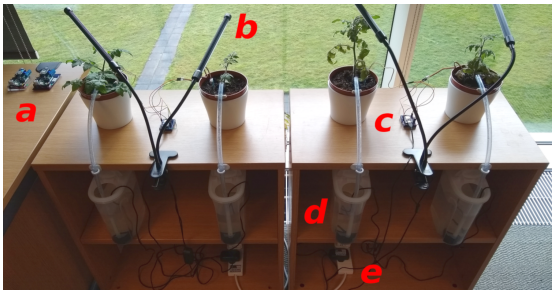

Fig. 6. Our indoor agriculture system.

VI. Evaluation

We evaluated CPoF on an indoor, automated agriculture system (Figure 6). This was motivated by the growing trend of *smart farms* that grow produce indoors, where environmental factors can be controlled to ensure a correct and efficient amount of light and water for produce [38], [39]. The objective of our evaluation is to see whether CEP systems are effective in providing FT support to handle data loss and inconsistency errors using the checks from Section IV-B.

This is distinct from the objectives of control engineering, which tries to control and maintain system state. Our contribution is a framework that identifies transitions into erroneous system states, and provides several established means of returning to an error-free state, such as: returning to the original state (*backward error recovery*); entering a new state (*forward error recovery*); or providing only a subset of the system services (*graceful degradation*). Our framework is designed to be applicable beyond our smart farm application, which is simply a test-bed to evaluate CPoF.

A. Case Study

Our system had two shelves, each with two plants. Beneath were four water containers (Figure 6d) that pumped water to their respective plants. We had grow lights above each plant (Figure 6b) that turned on when the room was dark, and off when bright. Each shelf had a microcontroller (Figure 6c) that had two moisture sensors placed into the soil of the plants, and a *light-dependent resistor* (LDR). Data were sent every 5 seconds to a Raspberry Pi 3 at the network edge.

There was two multi-sensors, *BoardWindow* and *BoardBackup* (Figure 6a), that collected infrared-light data, which was used to control the grow lights. The water pumps and grow lights were controlled using smart plugs (Figure 6e) that switched `ON`, `OFF`, or activated using a `TIMER` that switched it `ON` for t seconds then `OFF`. When a moisture value dropped below 0.5, it would trigger its corresponding water pump to activate using the `TIMER` function for 3 seconds. Similarly, if an infrared value from a multi-sensor was below 0.2, the grow lights would switch `ON`, or `OFF` if greater.

In our evaluation, we considered how the system detected and recovered from two scenarios: (1) when attempting to water a plant with no water left to pump; and (2) when BoardWindow suffered from a 'performance drop' i.e. a reduced

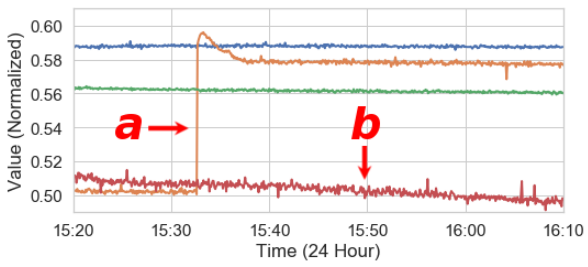

Fig. 7. Data from the four moisture sensors, represented by different colors.

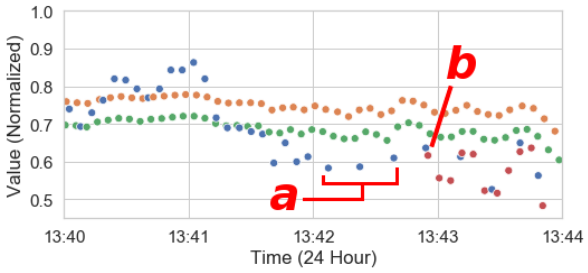

Fig. 8. LDR data from the two microcontrollers (orange, green), and infrared data from BoardWindow (blue) and BoardBackup (red).

data-transmission rate. Our methodology was to consider: (1) the applicability of the scenario with regard to the VFF framework; (2) the error(s) that might have propagated if the underlying fault were activated; and (3) the error-detection checks that would be needed to detect and recover from it. We used FlinkCEP v1.4.2[1] for our CEP implementation, and all data was normalized to the range $[0, 1]$.

B. Experiment: Empty Water Container

In this experiment, we identified a service failure whereby a plant was unable to be watered because its water tank was empty. Using VFF, we identified this as: an *environment* vulnerability, exploited by an *interaction* fault, that might have led to a *state transition* response failure. This was because, if the soil were not watered, the subsequent moisture data would not change due to a lack of state change in the physical world.

We first needed an NFA to identify a trend in moisture data. When soil was watered, the slope between the latest two moisture values should have become very large for a short period before stabilizing. We used the NFA from Figure 2b to first consume an initial moisture data event e_1. Subsequent moisture events were checked within a time window of 30 seconds. If any second event e_m in this time produced a slope ≥ 0.05, an error-detection event d_t was produced. We calculated the slope as $(e_m^v - e_1^v)/2$.

We implemented another NFA that checked if there was *not* a trend check within 30 seconds of `TIMER` occurring. We used the reversal check NFA (Figure 4), as follows. Event e_1 was fulfilled when the `TIMER` action was successfully executed. If 3 subsequent moisture events were received, with no d_t event in that time, it was assumed that no trend would occur. The run halted if d_t occurred.

In this experiment, the water container connected to the rightmost plant was empty. Its moisture data dropped to value 0.49 at approximately 15:49 (Figure 7b). This surpassed the 0.5 threshold and caused a `TIMER` action but failed to pump any water, as is reflected in the lack of trend change in the data. After 3 additional moisture values were received without a trend check event occurring, the reversal check NFA was fulfilled, indicating no trend after the `TIMER` action. The recovery strategy was to send an alert message to prompt human maintenance to resolve the issue. For comparison, the data from the center-right moisture sensor dropped below value 0.5 at approximately 15:32 (Figure 7a), causing a large trend increase shortly thereafter.

[1]https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/libs/cep.html

C. Experiment: Data Transmission Degradation

In this experiment, we identified a service failure whereby the data transmission rate from BoardWindow started to decrease i.e. performance degradation. Using VFF, we identified this as: a *network* vulnerability, exploited by an *interaction* fault, that might have led to a *timing* failure.

We first defined an NFA to identify when a drop in performance had occurred. Using the NFA from Figure 3a, e_1 represented an infrared-light data event from BoardWindow. When the next consecutive infrared light event was received, e_2, the difference between timestamps was checked to see if the time difference was > 10 seconds. We chose this threshold as it was double the 5 second data rate of BoardWindow. If the difference between events were ≤ 10, the run halted. This NFA produced error-detection event d_e.

We did not provide any error recovery for a single performance drop, as it might have simply been a isolated, transient error. Instead, we defined a second NFA to identify intermittent performance drops. Using the NFA from Figure 3b, if d_e occurred 3 times within 60 seconds, then the persistence of these errors led to new error events d_p, r_p. Our recovery was to activate and switch over to BoardBackup, to try to return the overall rate of infrared light data to ≤ 5 seconds. This can be seen in the live infrared data as it became intermittent at 13:42:10 (Figure 8a). Each of BoardWindow's data events took approximately 10/15 seconds to arrive. The wide intervals between these events caused d_e errors after each one. When this occurred 3 times in 60 seconds, an intermittent performance error was detected and caused d_p, r_p events, leading to the introduction of BoardBackup data (Figure 8b).

VII. Conclusion and Future Work

FT support is a key challenge for ensuring dependable IoT systems, with many existing implementations being static, tightly coupled, and inflexible. We proposed CPoF, where error detection was defined as NFAs and implemented in CEP systems to promote modularity and reusability in FT-support design. We prescribed the VFF framework for the categorical

design of error-detection NFAs, and used an automated agriculture system to demonstrate them. Using CPoF, we identified unreasonable environmental conditions and performance degradation via sensor data analysis.

In future work, we will expand our NFA taxonomy to all combinations in the VFF framework. This will help to establish a generic CEP framework for the easy and adaptive implementation of FT support in IoT systems.

REFERENCES

[1] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the internet of things: A survey," *IEEE communications surveys & tutorials*, vol. 16, no. 1, pp. 414–454, 2014.

[2] V. Bhuvaneswari and R. Porkodi, "The internet of things (iot) applications and communication enabling technology standards: An overview," *2014 International Conference on Intelligent Computing Applications (ICICA)*, pp. 324–329, 2014.

[3] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for internet of things: a survey," *IEEE Internet of Things Journal*, vol. 3, no. 1, pp. 70–95, 2016.

[4] E. Ojie and E. Pereira, "Exploring dependability issues in iot applications," in *Proceedings of the Second International Conference on Internet of things and Cloud Computing*. ACM, 2017, p. 123.

[5] I. Sommerville, *Software Engineering, Global Edition*, 10th ed. Pearson Education Limited, 2016.

[6] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*, ser. Dependable Computing and Fault-Tolerant Systems. Springer Vienna, 2012.

[7] T. N. Gia, A.-M. Rahmani, T. Westerlund, P. Liljeberg, and H. Tenhunen, "Fault tolerant and scalable iot-based architecture for health monitoring," in *Sensors Applications Symposium (SAS), 2015 IEEE*. IEEE, 2015, pp. 1–6.

[8] M. W. Woo, J. Lee, and K. Park, "A reliable iot system for personal healthcare devices," *Future Generation Computer Systems*, vol. 78, pp. 626–640, 2018.

[9] P. H. Su, C.-S. Shih, J. Y.-J. Hsu, K.-J. Lin, and Y.-C. Wang, "Decentralized fault tolerance mechanism for intelligent iot/m2m middleware," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*. Seoul: IEEE, 2014, pp. 45–50.

[10] A. La Marra, F. Martinelli, P. Mori, and A. Saracino, "Implementing usage control in internet of things: a smart home use case," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 1056–1063.

[11] S. A. Karthikeya, J. K. Vijeth, and C. S. R. Murthy, "Leveraging solution-specific gateways for cost-effective and fault-tolerant iot networking," in *2016 IEEE Wireless Communications and Networking Conference (WCNC)*. Doha: IEEE, 2016.

[12] Y.-L. Hu, Y.-Y. Cho, W.-B. Su, D. S. Wei, Y. Huang, J.-L. Chen, I.-Y. Chen, and S.-Y. Kuo, "A programming framework for implementing fault-tolerant mechanism in iot applications," in *Algorithms and Architectures for Parallel Processing*, G. Wang, A. Zomaya, G. Martinez, and K. Li, Eds. Zhangjiajie: Springer, 2015, pp. 771–784.

[13] A. Medvedev, A. Zaslavsky, M. Indrawan-Santiago, P. D. Haghighi, and A. Hassani, "Storing and indexing iot context for smart city applications," in *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, O. Galinina, S. Balandin, and Y. Koucheryavy, Eds. Cham: Springer International Publishing, 2016, pp. 115–128.

[14] G. Cugola and A. Margara, *The Complex Event Processing Paradigm*. Cham: Springer International Publishing, 2015, pp. 113–133.

[15] A. Buchmann and B. Koldehofe, "Complex event processing," *IT-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, vol. 51, no. 5, pp. 241–242, 2009.

[16] I. Flouris, N. Giatrakos, A. Deligiannakis, M. Garofalakis, M. Kamp, and M. Mock, "Issues in complex event processing: Status and prospects in the big data era," *Journal of Systems and Software*, vol. 127, pp. 217–236, 2017.

[17] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 15, 2012.

[18] H. Zhang, Y. Diao, and N. Immerman, "On complexity and optimization of expensive queries in complex event processing," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 217–228.

[19] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a better understanding of context and context-awareness," in *International symposium on handheld and ubiquitous computing*. Springer, 1999, pp. 304–307.

[20] S. Hasan, E. Curry, M. Banduk, and S. O'Riain, "Toward situation awareness for the semantic sensor web: Complex event processing with dynamic linked data enrichment," in *Proceedings of the 4th International Conference on Semantic Sensor Networks-Volume 839*. CEUR-WS. org, 2011, pp. 69–82.

[21] C. Barbero, P. Dal Zovo, and B. Gobbi, "A flexible context aware reasoning approach for iot applications," in *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, vol. 1. IEEE, 2011, pp. 266–275.

[22] D. Nallaperuma, D. De Silva, D. Alahakoon, and X. Yu, "A cognitive data stream mining technique for context-aware iot systems," in *Industrial Electronics Society, IECON 2017-43rd Annual Conference of the IEEE*. IEEE, 2017, pp. 4777–4782.

[23] A. I. Maarala, X. Su, and J. Riekki, "Semantic reasoning for context-aware internet of things applications," *IEEE Internet of Things Journal*, vol. 4, no. 2, pp. 461–473, 2017.

[24] Y. Liu, Y. Yang, X. Lv, and L. Wang, "A self-learning sensor fault detection framework for industry monitoring iot," *Mathematical problems in engineering*, vol. 2013, 2013.

[25] P. K. Choubey, S. Pateria, A. Saxena, V. P. C. SB, K. K. Jha, and S. B. PM, "Power efficient, bandwidth optimized and fault tolerant sensor management for iot in smart home," in *2015 IEEE International Advance Computing Conference (IACC)*. Bangalore: IEEE, 2015, pp. 366–370.

[26] A. Javed, K. Heljanko, A. Buda, and K. Främling, "Cefiot: A fault-tolerant iot architecture for edge and cloud," in *Internet of Things (WF-IoT), 2018 IEEE 4th World Forum on*. IEEE, 2018, pp. 813–818.

[27] K. F. Rauscher, R. E. Krock, and J. P. Runyon, "Eight ingredients of communications infrastructure: A systematic and comprehensive framework for enhancing network reliability and security," *Bell Labs Technical Journal*, vol. 11, no. 3, pp. 73–81, 2006.

[28] R. Isermann, *Fault-Diagnosis Systems: An Introduction from Fault Detection to Fault Tolerance*. Springer Berlin Heidelberg, 2006.

[29] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[30] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[31] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[32] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, 1991.

[33] I. Koren and C. M. Krishna, "Preliminaries," in *Fault-Tolerant Systems*. Morgan Kaufmann, 2010, ch. 1, p. 400.

[34] E. Bauer, *Design for Reliability: Information and Computer-Based Systems*. John Wiley & Sons, 2011.

[35] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch, "Distributed complex event processing with query rewriting," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. ACM, 2009, p. 4.

[36] L. L. Pullum, *Software Fault Tolerance Techniques and Implementation*, ser. Artech House computing library. Artech House, 2001.

[37] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic, "Stream reasoning and complex event processing in etalis," *Semantic Web*, vol. 3, no. 4, pp. 397–407, 2012.

[38] M. I. H. bin Ismail and N. M. Thamrin, "Iot implementation for indoor vertical farming watering system," in *Electrical, Electronics and System Engineering (ICEESE), 2017 International Conference on*. IEEE, 2017, pp. 89–94.

[39] J. Bauer and N. Aschenbruck, "Design and implementation of an agricultural monitoring system for smart farming," in *IoT Vertical and Topical Summit on Agriculture-Tuscany (IOT Tuscany), 2018*. IEEE, 2018, pp. 1–6.