Deriving a Unified Fault Taxonomy for Event-Based Systems

Waldemar Hummer Christian Inzinger Philipp Leitner
Benjamin Satzger Schahram Dustdar

Distributed Systems Group
Vienna University of Technology
Vienna, Austria
{lastname}@infosys.tuwien.ac.at

ABSTRACT

Dependability and fault-tolerance, which are key requirements for business- or safety-critical applications, require explicit knowledge of potential faults that may occur within a system. In contrast to other major research directions, the emerging field of distributed event-based systems is yet lacking a common understanding of faults. In this paper we take a step forward and study potential origins and effects of faults in such systems. Our work on a unified fault taxonomy follows a rigorous methodology. We first identify five core sub-areas in the broader field of event-based systems, and discuss commonalities and differences among them. Then we derive from the existing literature a coherent domain model, which accurately captures the specifics of the different areas. The domain model provides a holistic view and covers both structural and procedural aspects of event-based systems. Based on this model, we elaborate a detailed taxonomy of faults, in line with well-established fault dimensions from dependable and secure computing. The fault taxonomy forms the basis for a comprehensive discussion of fault instances across the five sub-areas of event processing.

Categories and Subject Descriptors

H.3.5 [**Online Information Services**]: Web-based Services; D.4.5 [**Operating Systems**]: Reliability; D.4.7 [**Operating Systems**]: Organization and Design; H.3.3 [**Information Search and Retrieval**]: Retrieval models

General Terms

Design, Management, Reliability, Standardization

Keywords

Event-based systems, fault taxonomy, dependable systems, event-driven interactions, stream processing, complex event processing, sensor networks, business processes

DEBS '12, July 16–20, 2012, Berlin, Germany.
Copyright 2012 ACM 978-1-4503-1315-5 ...$10.00.

1. INTRODUCTION

Event-based systems [21, 42, 51] in various fashions are gaining considerable momentum as a means for encoding complex business logic on the basis of correlated, temporally decoupled event messages. Distributed computing systems – including event-based systems – are often burdened with stringent requirements concerning dependability [37] and fault-tolerance [32], dictated by business regulations, contractual agreements, or laws. Fault-tolerance in a wider sense involves different aspects, such as fault detection, isolation, or recovery. Consideration of these aspects in the software development and validation process requires precise knowledge about the type and nature of faults that may potentially occur.

Many research directions have acknowledged the importance of a common understanding of faults. This has led to comprehensive surveys and taxonomies for very specific types of faults, including program and operating system faults [3], faults in object-oriented software [10], faults in service-based applications [11, 14], faults in memory devices [70], or faults in microprocessor design [4]. Event processing, as an emerging research direction, is yet lacking a common model for faults. Existing works on faults in distributed systems are partly applicable as well, but some characteristics that are common among event-based systems (e.g., event correlation or timing aspects) call for a more specific analysis and classification.

In this paper, the main contribution is an initial fault taxonomy for event-based systems. We study different aspects of faults in event-based systems in general, with a particular focus on five main sub-areas: (1) event-driven interaction paradigms (EDIP) [13, 22, 23, 33, 53], (2) event stream processing (ESP) [1, 6, 7, 9, 15, 64], (3) complex event processing (CEP) [21, 42, 75], (4) event-driven monitoring networks and wireless sensor networks (WSN) [2, 12, 35, 46, 52, 54, 58], and (5) event-driven business process management (EDBPM) [57, 59, 69, 71], including the related field of event-driven service-based systems [49, 74].

We identify various requirements from these areas, derive a common (unified) event processing model and discuss potential faults in these systems along different dimensions. Note that the distinction between the above five types of event-based systems is not always clear-cut. Contrarily, the five areas share many common concepts and characteristics, which leads to the idea that the rich body of knowledge from different fields should be combined into a unified taxonomy.

1.1 Methodology

Deriving and applying a fault taxonomy requires a rigorous methodological approach, which is briefly describe here.

We first establish a common model for distributed event-based systems, which captures general properties across different types of event-based systems. To that end, we identified more specific sub-areas in the broad field of event processing. The model is a contribution in itself, as it summarizes artifacts, properties and requirements of different event processing disciplines and provides a guideline how to integrate these artifacts into a single reference model. Similar efforts have been undertaken in the past (e.g., [50, 73]) and the goal is to find commonalities among the existing approaches. What distinguishes our model from existing approaches is that we specifically seek potential origins of faults. Hence, we focus on the event processing artifacts that we deem most relevant, as well as connections, interactions and relations between them.

Next, the core concepts and terminology are clarified and applied to our context. Among other things, we recite the often used distinction between failures, errors and faults [5] and discuss their relevance in our context. We then define the dimensions along which the fault taxonomy is established. Concerning the taxonomy dimensions, we distinguish between fault classes on the one hand, and fault sources on the other hand. Afterwards, we combine the sub-areas and dimensions and discuss types and manifestations of possible faults in event-based systems. Where applicable, the fault description refers back to the affected artifacts in the model.

Although not directly a part of this paper's contribution, we also briefly outline in the conclusions how the obtained model and fault taxonomy can be implemented and applied to real-life platforms with the goal of fault injection [31, 32] and fault diagnosis [30], which is our focus for future work.

1.2 Roadmap

The remainder of this paper is structured as follows. Section 2 defines the common model for event-based systems, which is used throughout the paper. Section 3 constitutes the core contribution where we work out selected classification dimensions, establish the fault taxonomy along these dimensions, and provide an in-depth discussion of fault instances for each of the five discussed sub-areas. Related work that deserves special consideration and has not been mentioned in the main part of the paper, is discussed in Section 4. Finally, Section 5 concludes the paper, discusses remaining limitations, and points to application scenarios and future research directions.

2. A COMMON MODEL OF DISTRIBUTED EVENT-BASED SYSTEMS

In this section, we establish a common model of event-based systems, which serves as the basis for discussion in the remainder of the paper. The goal of the model is to capture specifics of different variants of distributed event-based systems. In particular, the model is derived from various previous publications in five sub-areas, which we briefly discuss in Section 2.1. The challenge in defining such a reference model is the tradeoff of including as many aspects and different viewpoints as possible, while at the same time keeping the complexity at a minimum, providing the necessary level of generality.

2.1 Specializations

We have evaluated various publications published as books, journal or conference contributions related to event-based systems, and have extracted five main sub-areas of this field.

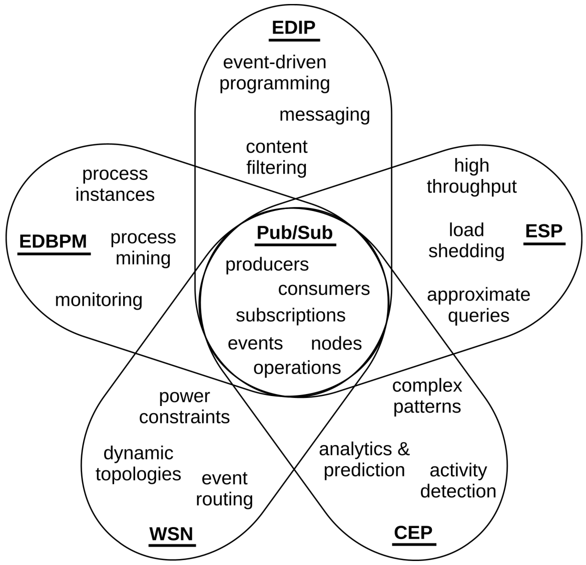


Figure 1: Sub-Areas of Event-Based Systems

The first principal field involves event-based information dissemination [44, 53] and event-driven programming models [17], including content filtering [33], message oriented middleware (MOM) [8], wide-area notification services [13], message-passing systems [20], or tuple spaces [26]. Examples where this field plays a key role are emergency control systems, real-time collaboration, or active databases [47]. Here we collectively refer to this class of systems as **event-driven interaction paradigms** (EDIP). The event-based interaction mode has also gained importance in software engineering, e.g., for graphical user interface software [65], in the context of specification of system architecture [41, 45], or under the term *implicit invocation* [25].

The second field is **event stream processing** [6, 7, 64] (ESP), which deals with continuous queries over data streams, often with a focus on high-frequency events and scalability. Example applications are financial services [9], stock trading platforms [1], or network traffic management [7].

The third field is **complex event processing** [21, 42, 43] (CEP), which covers the core concepts of causal event histories, event patterns, event filtering and event aggregation [43]. Application areas include geospatial event processing [21], RFID-based product monitoring [72], or online fraud detection [21, 60].

The fourth main area of interest is event-driven **wireless sensor networks** [2, 12, 58] (WSN), including applications of ubiquitous computing [66], intrusion detection [35], or monitoring of environment (temperature) data [58]. Among the key problems in this field are energy-efficiency, event routing or data aggregation [34].

The fifth area is **service-oriented and event-driven business process management** [69, 74] (EDBPM), with popular examples including road tolling systems [21], order and shipment management [57], or workflows of telecommunication providers [49].

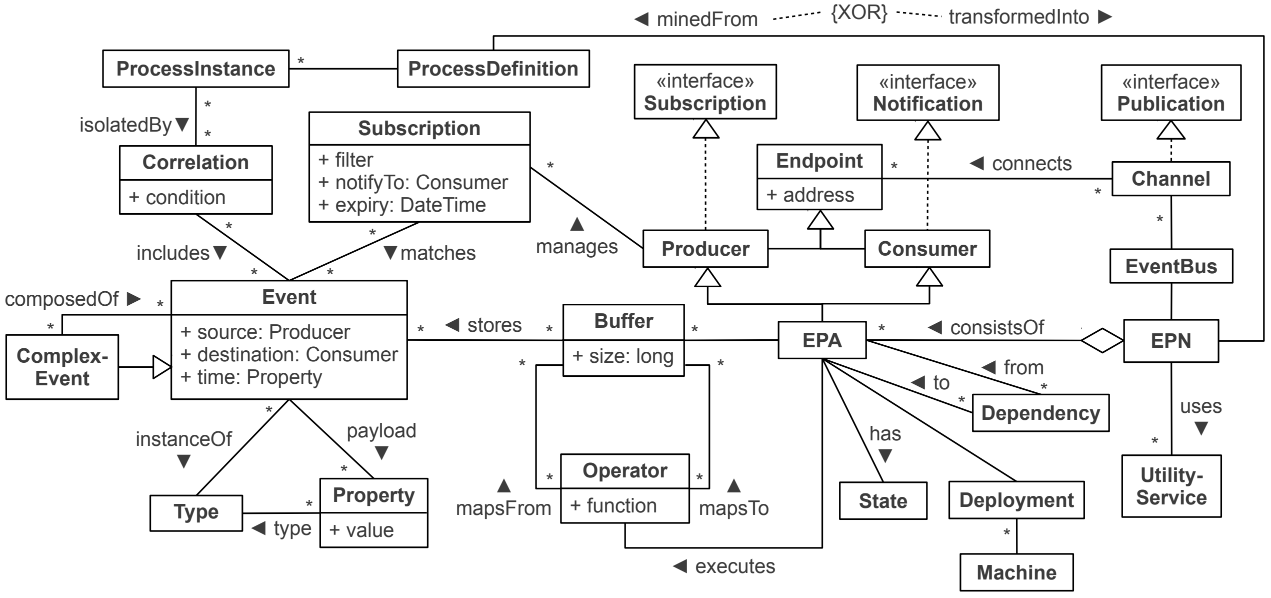


Figure 2: Excerpt of the Common Model for Event-Based Systems

Figure 1 contains five elliptic shapes which represent the five mentioned event processing areas. The intersection of the shapes is a circle that contains concepts of the **publish/-subscribe** (Pub/Sub) interaction scheme [22, 23]. Pub/-Sub elements are commonly found in all areas (e.g., events, producers, consumers), although sometimes slightly different terminology is used in each field. Moreover, there are characteristics and challenges specific to each area, which are printed in the non-intersecting parts of the figure. To further illustrate some of the commonalities among the five sub-areas, Table 1 lists typical terminology in the different areas which often refers to similar concepts.

The concept of events plays a role in many other research areas that are rather remotely related to our choice, for instance interconnect solutions for large scale multiprocessor systems [16], discrete event systems [55], or events/signals in operating systems [28]. Although partly applicable to our approach, these areas are not explicitly included in the discussion of this paper.

2.2 Description of the Common Model

In the following we discuss the unified model for event-based systems. The model we propose is based on previous work that has tried to capture common features of event-based systems and applications, most notably in [50, 62, 73]. The fundamental concepts, artifacts and entities of the model (printed in **bold**) are introduced below. The core elements and relationships of the model are depicted in the form of a UML class diagram in Figure 2.

- An **event** is "*anything significant that happens or is contemplated as happening*" [50]. Research distinguishes **simple events** (events which do not represent a set of other events) and **complex events** (events that span multiple other events). An event has a certain **type** and an arbitrary number of **properties**. Standard properties are the **source**, **destination** and the **time** at which the event occurred. Additionally, the event may be associated with application-specific properties, denoted as event **payload**.
- Events are typically sent from a **producer** (often termed **source** in sensor networks) to one or more **consumers** (**sink** in sensor networks) through a communication **channel**. Between the event producer(s) and end-consumer(s) there is an **event processing network** (EPN) consisting of **event processing agents** (EPAs) connected by event channels. The channel may be a direct connection (i.e., the producer is able to contact the consumer(s) directly), or it may be part of an **event bus** whose responsibility is to deliver the events accordingly. Typically, additional functionalities and responsibilities are attributed to the event bus,

Concept [50]	EDIP	ESP	CEP	WSN	EDBPM
event	notification	tuple	event	datum	invocation
producer	publisher	source	producer	sensor	service/ activity
consumer	subscriber	sink	consumer	sink	
event processing	service	operator	agent	node	
channel	channel	stream	event bus	link	service bus
derived event	merged message	event pattern	complex event	fused information	composite service

Table 1: Different Terminology for Similar Concepts

such as storage, registry or access control services [50]. For simplicity, we summarize these artifacts under the term **utility services**, which the EPN interacts with. Another critical utility service is **time synchronization**, required for correct and consistent timestamping of events. In stream processing, EPNs and data flow dependencies between EPAs are often modeled as a directed **acyclic** graph (DAG). However, in general the connections between EPAs may also be **cyclic**.

- Event consumers register a **subscription** with a producer to receive **notifications** about certain events. The subscription **filter** specifies which events are of interest, based on the type and/or payload. Upon receipt of an event, consumers **react** by performing an (application-dependent) **task**, e.g., operating a physical switch, invoking an electronic service, initiating a business process etc. Subscriptions must be unambiguously **identifiable**, and consumers should be **addressable**. If notifications cannot be directly **pushed** to consumers (via a **publishing interface**), they have to be requested in **pull** mode. Internally, event producers may **bundle** subscriptions of different consumers that are similar/equal from a processing perspective.
- **Correlation** [35, 57] means identifying and grouping events that are logically linked together (from the application point of view). A correlation **condition** is a function that determines for a set of events whether or not they are correlated. In the simplest case, **correlation properties** are defined on two or more event types and all instances of these types for which the properties **match** are considered correlated [57]. Event **isolation** aims at dividing the total set of events into (usually disjoint) sets of correlated events (a typical example is isolation of business process instances).
- The logic of an event-based business process is specified in a **process definition** (e.g., using a graphical notation or some formalization like *Petri nets*) of which multiple **instances** can exist. The process definition is either known a priori and **transformed** into an EPN, or the definition is learned by monitoring the EPN using **process mining** techniques [57, 69].
- Communication between EPAs often (although not necessarily) happens **asynchronously** and **non-blocking**. Events that cannot be processed immediately are put to a **buffer** (or queue). Buffers are subject to physical resource restrictions and therefore usually have a **limited size** (length).
- Event **routing** may happen either **statically** (according to pre-defined routing tables) or **dynamically** (decision based on the characteristics or capabilities of available EPAs). Dynamic (and resource efficient) routing is a key problem in WSNs [12, 34].
- EPAs process a set of input events and output zero or more (possibly new) events. Thereby, three stages are distinguished [50]: **pattern matching**, **processing** and **emission**. An EPA can be at the same time event consumer and producer, and hence it implements the corresponding interfaces. Incoming events are put to one or possibly multiple (depending on the implementation) **input buffers**. The **operator** of an EPA is responsible for generating events on the **output buffers** and is specified via a **function** that maps from inputs to desired outputs (see details in Section 2.3). Additionally, each EPA is associated with a **state** that reflects its current allocation of variables, memory, registers, and other state information. We assume that the state also includes the model instances that are relevant for the EPA, so that the EPA can **reflect** on itself at runtime (e.g., which subscriptions it manages). Apart from the general model entities stored in the state, the notion of state is highly application specific. Hence, we only assume this generic container and make no further restrictions about its representation, in order to keep things simple.
- EPAs are **deployed** on physical **machines** (or computing nodes), and one machine can **host** multiple EPAs. If the deployment changes and the responsibility of hosting an EPA (including its state) is transferred from one machine to another, we speak of **migration** of this EPA.

2.3 Modeling the Operation of EPAs

Because a large part of an EPN's functionality is encoded in the EPAs, they are a primary source for potential faults. Hence, we discuss the internal structure and *modus operandi* of EPAs, as envisioned in the common model, in more detail.

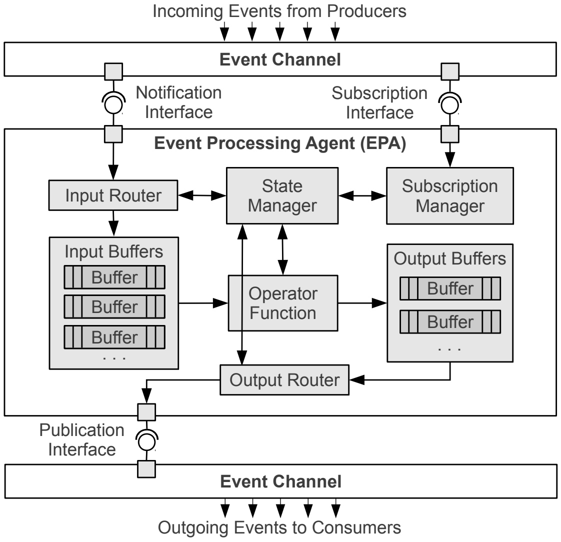


Figure 3: Internal Structure And Functionality Commonly Applicable to EPAs

Figure 3 depicts a UML component diagram with an EPA connected to two channels. The state associated with the EPA is maintained by a **state manager**, which may be implemented as an actual state machine (automaton) or some other mathematical model. The **input router** is responsible for directing the events received via the **notification interface** to one or more input buffers. The EPA also receives requests from the channel via the **subscription interface**. A **subscription manager** is responsible for maintaining subscriptions. More specifically, when a new request comes in, a new *Subscription* model element is instantiated and stored in the state manager. The **output router** is responsible for forwarding events from the **output buffer** to subscribed consumers via the channel's **publication interface**. The figure contains five exemplary buffers for illustration, but the dots ("...") indicate that more buffers are conceivable.

2.3.1 Input-Output Operator Function

The **operator function** mediates between the input/output buffers and the state manager. Formally, this function is defined as follows. Let B denote the set of buffers (for both input and output), E the set of all events, T the temporal domain (all possible event timestamps), and S the set of possible states. The current content of a buffer is expressed as $\mathbb{N} \to E$, mapping from the numeric slot position (index) within the buffer to an event ($\varnothing$, the "empty event", is also part of E and hence $\varnothing$ signifies an empty buffer slot).

$$\phi : [T \to E]^n \to [T \to E]^m; n, m \in \mathbb{N}^+ \quad (1)$$

A generic operator function ϕ, denoted *stream transformer*, has been previously proposed in [64]. This function (printed in Equation 1) takes a set of timestamp/event pairs as input and outputs a new set of timestamp/event pairs.

$$op : (BE \times S) \to (BE \times S) \quad (2)$$

We propose to use a more specific input-output operator function that considers the input/output buffers as well as the state of the EPA. Let $BE := \mathcal{P}(B \times (\mathbb{N} \to E))$ denote the buffered events, i.e., the buffer allocation at any point in time ($\mathcal{P}(x)$ denotes the *powerset* of x). We then define the operator function as printed in Equation 2. The input of the *op* function is a subset of the buffers (BE) together with their content, plus the current state (S) of the EPA. The output of *op* is a new content assignment for a subset of the buffers (BE), plus a new assignment for the EPA state (S).

Upon arrival, new events are added to the corresponding input buffer(s) (existing events are shifted by one position), and *op* is executed. The approach provides the expressive power of *event-condition-action* (ECA) rules [47], a popular method for CEP specification. Our formalization is also in line with the stream transformer definition in [64], since the timestamp is accessible as an event property in our model.

We illustrate the operator function with a small example in Equation 3. The example considers an EPA which receives numeric event values on two input buffers (ib_1, ib_2) and determines whether the sum of the values is positive or negative. Let $val(e)$ denote the numeric payload of an event $e \in E$. The EPA has two output buffers (ob_1, ob_2). If the sum is positive an event e_{pos} is put to ob_1, otherwise the new event e_{neg} on ob_2 indicates that the sum is negative. Additionally, the EPA can be in a state $INACTIVE$, in which case no output is generated at all.

$$op(\{(ib_1, \{1 \mapsto e_1\}), (ib_2, \{1 \mapsto e_2\})\}, s) :=$$
$$\begin{cases} (\emptyset, s) & \text{if } s = INACTIVE \\ (\{(ob_1, \{1 \mapsto e_{pos}\}), s) & \text{else if } val(e_1) + val(e_2) \geq 0 \\ (\{(ob_2, \{1 \mapsto e_{neg}\}), s) & \text{else if } val(e_1) + val(e_2) < 0 \end{cases} \quad (3)$$

2.3.2 Event Routing Functions

Particularly in WSNs, dynamic event routing is a key challenge [12, 34]. Our model, therefore, contains two router components, which reflect that routing is decoupled from the input/output operator. Let P denote the set of event producers and C the set of event consumers. The input router is defined via a function $in : (E \times P \times S) \to \mathcal{P}(B)$, which determines for an incoming event $e \in E$, producer $p \in P$, and current state $s \in S$ the subset of input buffers $ib \subseteq B$ to which e is added. Conversely, the output router is defined via a function $out : (E \times \mathcal{P}(B) \times S) \to \mathcal{P}(C)$, which defines for an outgoing event $e \in E$, coming from a subset of the output buffers $ob \subseteq B$, and a current state $s \in S$ the consumers $c \subseteq C$ to which e will be forwarded.

3. FAULT TAXONOMY

Based on the model defined in Section 2, we now establish the fault taxonomy for distributed event based systems.

3.1 Taxonomy Dimensions and Terminology

The highly influential work by Avizienis *et al.* [5] studies concepts for dependable and secure computing, and provides a detailed taxonomy framework with multiple fault dimensions. Despite the high level of detail, their work still provides general applicability and builds a solid basis for extended approaches. For instance, Chan *et al.* [14] have presented a fault taxonomy for Web Service Compositions that closely builds on the classification by Avizienis *et al.*

Firstly, we recite the most relevant terminology from [5], put into the context of event-based systems. A system delivers **correct service** if it provides the desired functionality, which includes the functionality of end producers and consumers as well as the EPN that mediates between them. A **failure** occurs when the system *"does not comply with the functional specification, or because this specification did not adequately describe the system function"* [5]. As an example, consider a system that analyzes a stream of stock market events and is supposed to indicate if the price of a stock "rises significantly", but no event is generated, even after the price has risen ten consecutive times. Depending on the system function, this behavior may either be a failure caused by incorrect processing, or the failure may be rooted in the fact that the system detects stock rises with a high statistical confidence of 99.9%, whereas the specification (implicitly) assumed a 95% confidence interval. When asking for the manifestation of a failure in the system, we say that a failure is caused by one or more states deviating from the correct service state. This deviation is denoted as **error**. The assumed cause of an error, either internal or external, is called a **fault**. In the stock price example, a possible error is that an EPA was unable to store new incoming events, and the probable fault that lead to this error is a buffer overflow. Note that not all faults cause an error and therefore lead to a system failure: *"A fault is **active** when it causes an error, otherwise it is **dormant**"* [5].

3.1.1 Fault Classes

In [5], 16 elementary fault classes are derived from eight basic viewpoints. We have identified 12 of these fault classes as highly relevant for our purpose. The *phase of creation or occurrence* distinguishes between faults that are introduced at development time or during operation (execution) of the system. *System boundaries* refers to the distinction whether a fault is caused internally within the system or caused by external input received at the service interface or from the environment. *Persistence* determines whether the fault is continuous our bounded in time (i.e., persistent or transient). The *dimension* indicates faults that affect (or originate in) either software or hardware. The *phenomenological cause* of a fault can be either rooted in natural phenomena (on which humans have limited or no influence) or in active human participation. The *capability* dimension acknowledges that some faults are introduced inadvertently (or accidentally), while other faults result from lack of professional competence, strategy or planning.

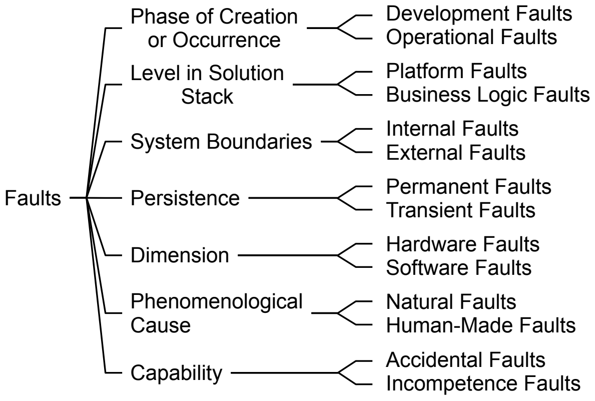


Figure 4: Elementary Fault Classes (based on [5])

The other four fault classes in [5] make a distinction between *malicious* and *non-malicious faults* as well as *deliberate* and *non-deliberate faults*). These four latter types of faults are particularly important in the area of security, which is not the core focus in this paper. Hence, our approach is to capture the technical manifestations of deliberate and malicious acts (e.g., an overloaded channel or buffer overflow caused by a denial-of-service attack), but our fault taxonomy does not explicitly distinguish purely security-related dimensions. In return, we add two important fault classes concerning the *level in solution stack*, namely *platform faults* versus *business logic faults*. The former class of faults has its roots in the implementation of the underlying event processing platform and may potentially affect all of the applications on top of it, whereas the latter fault class is tightly connected to the types of events and the specific business application that is deployed by the platform. Figure 4 contains a schematic view of the seven dimensions and 14 classes used in our fault taxonomy.

3.1.2 Fault Sources

Besides classes (types) of faults, our taxonomy also asks for the sources of faults, i.e., the artifacts of the system which are potentially or positively responsible for causing the fault. Figure 5 depicts the six categories of fault sources, which have been extracted and compiled from earlier work on fault localization [3, 63] and root cause analysis [38]. Where applicable, the fault source description refers back to elements of the model discussed in Section 2.

The *environment* refers to the physical platform on which the event-based system operates, i.e., machines, network links, and power supply. Power supply plays a key role, particularly if the event data are only stored in volatile (non-persistent) memory, which is flushed in case of a power outage. *External input* is received from business events and utility services; both types of inputs can potentially influence the reliable operation of the system. The fault source category named *code functions* refers to processing logic encoded as operators, state transitions, queries, algorithms, etc. Evidently, code functions are at the core of event processing and can introduce faults into the system. The *system state* includes both the model-related configuration of the system (e.g., active correlations, subscriptions) and application-specific business logic state (e.g., state $INACTIVE$ in the example in Section 2.3.1). *Software assets* are self-contained components that the system builds on, which are known to operate well under controlled conditions but fail under certain circumstances (e.g., buffer overflow of a channel, or kernel error of an operating system). The *concepts and abstractions* category captures additional aspects that play a role in the processing, such as timing aspects or dependencies. The third part of this category is denoted semantics; for instance, if an event-processing platform performs dynamic re-configuration and re-deployment, it must be ensured that the newly configured system is semantically equivalent to the previous state and still fulfills all requirements.

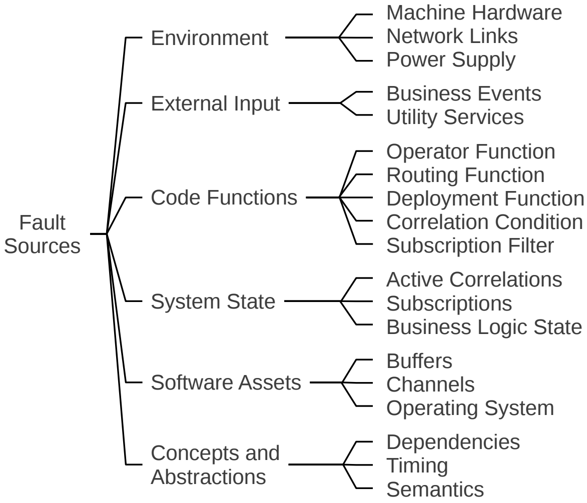


Figure 5: Fault Sources

3.2 Discussion of Identified Faults

In Figure 6, we identify and classify 30 fundamental faults (without claim of completeness), which are the result of literature review guided by combining different fault classes and model artifacts. The figure depicts a classification tree in which the leaf nodes are fault descriptions, and each level represents a pair of fault classes from Figure 4.

Moreover, Figure 7 lists for each of the 30 faults the sources which are likely involved in the creation of the fault. The matrix contains fault sources on the left-hand side and fault examples on the top. Each intersection of fault type and matching fault example is marked with a red dot. Note that a fault may potentially be rooted in more than one fault source. We have attributed each fault to one of the core sub-areas of event-based systems that we identified in Section 2.1 (see "Main Areas Affected" at the bottom of Figure 7). This is of course a strong simplification – in fact, the distinction is not always clear and the areas partly overlap, i.e., faults may play a role in more than one area.

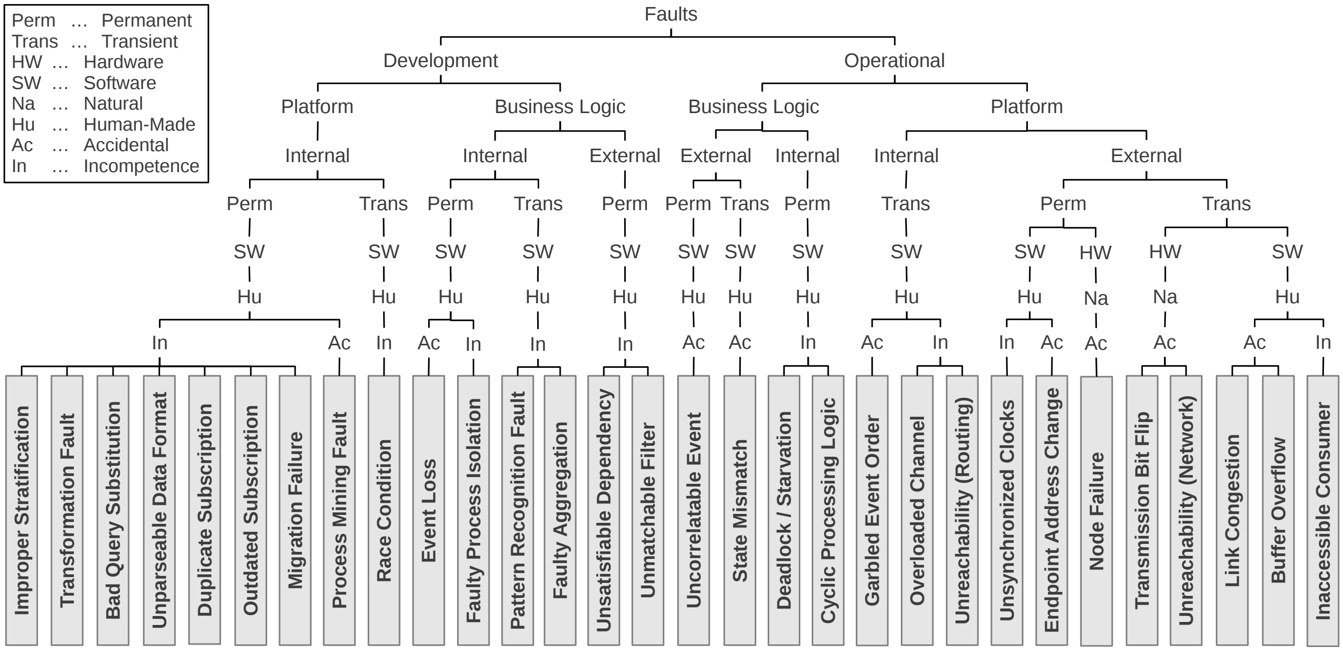


Figure 6: Taxonomy Tree for Faults in Event-Based Systems

3.2.1 *Faults in Pub/Sub Systems*

Publish/Subscribe is the basis for many types of event-based systems. The main challenges are related to management of subscriptions, filtering, and dissemination of information [22]. On the network level, multicast transmission between one producer and multiple subscribers is often achieved using point-to-point communication primitives. This may easily lead to **Overloaded Channels**, which is the first (leftmost) fault example in Figure 7. For illustration purposes we briefly explain the fault classes for this fault (see Figure 6). An overloaded channel is an *operational fault*, because its phase of creation/occurrence is attributed rather to execution time than to development time. It is not particularly business logic-specific, but a general *platform fault*. The fault usually comes into existence by *external* influences or inputs. Although it remains active for a while, its presence is bounded in time (until the traffic drops to an acceptable level) and hence we note that it is *transient*. Moreover, the fault is a *software fault* and it is *human-made*. Concerning the fault capability, a channel overload is considered *accidental*, especially if the underlying network is commonly used by multiple external systems.

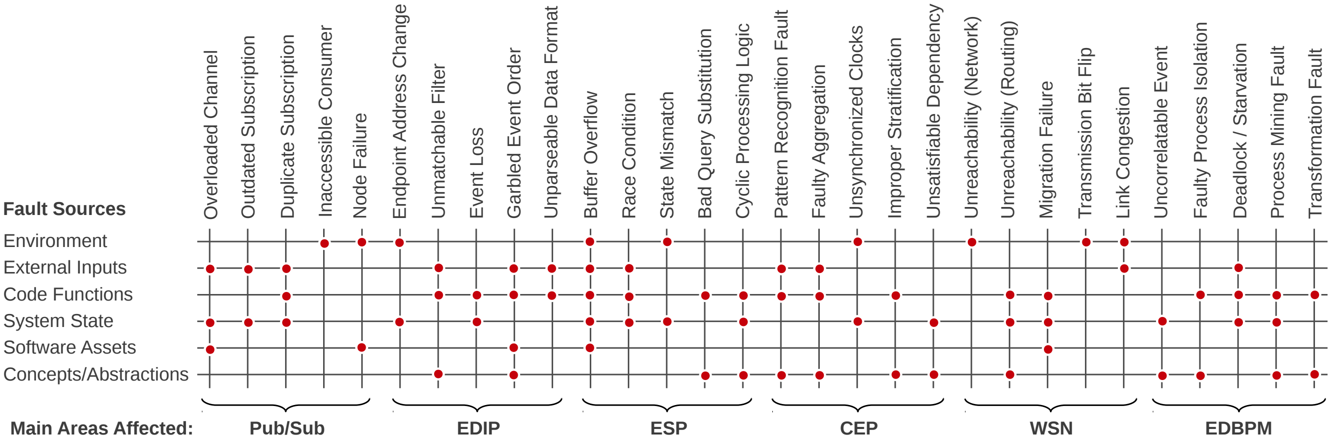
As subscriptions have an expiry time in our model, it may occur that an event producer garbage collects an expired subscription, while one of the consumers still retains a reference and attempts to modify it. This type of fault is denoted **Outdated Subscription** and can be compared to a *dangling pointer* in programming languages. This fault is generally attributed to the platform and the development phase, because outdated subscription references should be properly garbage collected. Analogously, a subscription should be kept alive as long as there exist any references to it. Similarly, a **Duplicate Subscription** is a development fault because one would expect that the platform takes care of eliminating such duplicates. Concerning delivery of event messages in push mode, an **Inaccessible Consumer** occurs if the endpoint address of the consumer is not available. This can have several reasons, e.g., the consumer process has no privileges to open a listening network socket, or network packets are dropped due to firewall rules, etc.

3.2.2 *Faults in Event-Driven Interactions*

Related to the Inaccessible Consumer fault in Pub/Sub is the problem of **Endpoint Address Change**, which we attributed to the EDIP category. This fault happens when the logical consumer of an event subscription is moved to a different physical machine or connection without updating its references. A main difference is that an address change is usually permanent, whereas an inaccessible consumer may be a permanent or temporary (transient) fault.

Another weak point in EDIP (and also Pub/Sub systems) is the evaluation of filters. Popular implementation variants include topic-based filtering and content-based filtering [22]. While topic-based filtering is usually straight-forward, content-based filtering is more dynamic and flexible. For instance, in [33] a content-based Pub/Sub system with routing based on *Bloom filters* has been proposed. This leaves room for optimizations in the dissemination procedure, but also opens possibilities for the introduction of new faults. We collectively refer to this class of problems related to evaluation of subscription filters as **Unmatchable Filter**.

Event Loss is a potential problem in most event-based systems. In contrast to load shedding [1,18], where strategies for deliberate dropping of events are put into action, event loss in CEP can also happen unintentionally. It is particularly relevant when an event is expected to be handed from a source to a destination EPA and the global behavior depends on the operator of each involved EPA along the path. A problem closely related to event loss is **Garbled Event Order**. For instance, the *Aurora* platform implements delayed processing in periods of high load, which may result



Fault Sources	Overloaded Channel	Outdated Subscription	Duplicate Subscription	Inaccessible Consumer	Node Failure
Environment				●	●
External Inputs	●	●	●		
Code Functions			●		
System State	●	●	●		
Software Assets	●				●
Concepts/Abstractions					
Main Areas Affected:	Pub/Sub				

Fault Sources	Endpoint Address Change	Unmatchable Filter	Event Loss	Garbled Event Order	Unparseable Data Format
Environment	●				
External Inputs		●		●	●
Code Functions		●	●	●	●
System State	●		●		
Software Assets				●	
Concepts/Abstractions		●		●	
Main Areas Affected:	EDIP				

Fault Sources	Buffer Overflow	Race Condition	State Mismatch	Bad Query Substitution	Cyclic Processing Logic
Environment	●		●		
External Inputs	●	●			
Code Functions	●	●		●	●
System State	●	●	●		●
Software Assets	●				
Concepts/Abstractions				●	●
Main Areas Affected:	ESP				

Fault Sources	Pattern Recognition Fault	Faulty Aggregation	Unsynchronized Clocks	Improper Stratification	Unsatisfiable Dependency
Environment			●		
External Inputs	●	●			
Code Functions	●	●		●	
System State			●		●
Software Assets					
Concepts/Abstractions	●	●		●	●
Main Areas Affected:	CEP				

Fault Sources	Unreachability (Network)	Unreachability (Routing)	Migration Failure	Transmission Bit Flip	Link Congestion
Environment	●			●	●
External Inputs					●
Code Functions		●	●		
System State		●	●		
Software Assets			●		
Concepts/Abstractions		●			
Main Areas Affected:	WSN				

Fault Sources	Uncorrelatable Event	Faulty Process Isolation	Deadlock / Starvation	Process Mining Fault	Transformation Fault
Environment					
External Inputs			●		
Code Functions		●	●	●	●
System State	●		●	●	
Software Assets					
Concepts/Abstractions	●	●		●	●
Main Areas Affected:	EDBPM				

Figure 7: Factors of Influence Responsible for Different Faults

in the situation that already emitted results need to be revised [1]. This feature is prone to errors and it is crucial to restore the correct message order. Applied to our model, garbled event order may also happen if the input router or output router functions are not correctly synchronized.

In some cases, event-driven interaction faults are less sophisticated but simply caused by an **Unparseable Data Format** of the underlying events. There exists a variety of different character encodings and document markup languages used to encode the payload (event properties), and the communication will most likely fail if the event producer and consumer do not utilize the same standards.

3.2.3 *Faults in Event Stream Processing*

We now focus on faults that are related to the main challenges in event stream processing (ESP) systems. An obvious and often studied source for problems is high-frequency streams. Resource limitations of the machines on which EPAs are deployed allow only a certain amount of events (or tuples, as often denoted in stream processing) to be stored and processed per time unit, depending on the complexity of the performed operation. A **Buffer Overflow** occurs if an EPA cannot allocate new memory to buffer an incoming event. This problem has been intensively studied and different solutions were proposed. One solution is load shedding [1, 18], where exactness/accuracy in the operator function is sacrificed for continuous availability by heuristically dropping (or delaying) events that are (deemed to be) of less importance. Partial fault tolerance under high load and the effects of bursty tuple loss have been intensively studied based on a fault injection framework in [31].

A **Node Failure** occurs when the event processing system experiences hardware failures or faults due to "heisenbugs" in the underlying operating system or platform [61]. This type of fault also plays a key role in WSNs, for instance caused by depletion of batteries. We assume that the effects of a node failure (e.g., loss of state information) are permanent and cause the EPA operator to stop functioning immediately. From the viewpoint of the event processing system, a node failure happens accidentally and cannot be directly attributed to human incompetence. Evidently, node failure poses a key problem to any type of system that is supposed to operate reliably. EPA replication is a commonly used technique for reliable operation in the presence of node failures, which is intensively studied in [61].

Another key issue with parallel processing of events (in particular for ESP, but also in other areas) is that improperly synchronized code can cause **Race Conditions** when common resources (e.g., buffer memory) are accessed. Synchronization errors may lead to unexpected side effects and inconsistent states. These problems are often hard to debug and only revealed nondeterministically under high load.

Reliable reflection about the system state plays a key role for the operator and routing function of EPAs. If the actual state of the system is not accurately reflected in the saved state of an EPA, we speak of a **State Mismatch**. Two main reasons may be responsible for a state mismatch: either the output of an EPA's operator function is faulty and generates a (locally) inaccurate state, or the state transfer of a global state between multiple EPAs is not transactionally safe.

Queries over streams that involve multiple nested operations are often decomposed and rewritten/optimized based on query plans to achieve high performance [1, 75]. The goal is to map the original query to a new query which has more desirable characteristics (e.g., better suited for distribution or avoids unnecessary execution steps), while at the same time preserving the exact semantics of the original query. If the latter condition does not hold, a **Bad Query Substitution** fault occurs that may have an effect on the system functionality.

Finally, if the processing graph defined by the dependencies within an EPN contains cycles, the system may get into a state in which events are "trapped" by getting forwarded indefinitely. We refer to faults of this type as **Cyclic Processing Logic**. A possible technique to avoid this fault is to attach metadata to event messages that circulate in the system, for instance a time-to-live counter that specifies how many times the event may be passed on to the next EPA.

3.2.4 *Faults in Complex Event Processing*

Complex Event Processing (CEP) is concerned with complex interactions and event patterns often spanning multiple sources. A complex event is one that is derived or aggregated from multiple other events. A frequently encountered problem are **Pattern Recognition Faults**. One manifestation of this fault is when a relevant pattern exists but is not detected. The other possibility is that a complex event (e.g., credit card fraud [60]) may be mistakenly assumed to exist although in fact it does not. Pattern recognition is often achieved using event automata [60] which define states and state transitions. In our model, event automata correspond to the EPA state combined with its operator function. Pattern recognition faults are hence rooted in the incorrect handling of the EPA's state by this function. The general case of events being incorrectly combined into complex (or aggregate) events is denoted as **Faulty Aggregation**. Aggregation functions, including simple aggregates (count, sum, average, minimum, maximum, etc.) or mathematical functions over event sequences (e.g., cross-correlation coefficients), are a vital foundation for defining and executing CEP business logic [60,72,75]. As indicated in Figure 6, faulty aggregation usually affects the business logic (e.g., when combining multiple correlated "item" events into a single "package" event in a warehouse [72]).

Time-sensitive CEP queries and event composition often rely on synchronized time measurements among the producers, to retain either absolute time difference or relative order of events [40]. In our model, time is expressed as a special case of the EPA state, which is periodically changed (irrespective of event inputs) and shared among the EPAs. **Unsynchronized Clocks** are therefore a source of failure which may cause misbehavior in a CEP system. Synchronization is also a key issue in wireless sensor networks [2].

Stratification [36] is the process of splitting up the EPN dependency graph into independent sub-graphs, denoted as stratums, to achieve parallelism and early filtering of events. The semantics and dependencies of the processing graph must be retained and hence the non-trivial stratification algorithm in [36] is itself a potential source of faults. The effect of **Improper Stratification** could be for instance that events are filtered out too early, or that the wanted effect of load distribution is not achieved and one node or channel gets overloaded.

CEP systems may run into the problem of an **Unsatisfiable Dependency**, where an EPA is in a state in which it expects a certain event to arrive but this event cannot be delivered, e.g., caused by event loss or a cyclic dependency. Note that we have to make a subtle distinction here: the processing graph of an EPN *can* in fact be cyclic (e.g., an EPA may consume and process events that were emitted by itself); however, the causal dependency of correlated or complex events must not be cyclic. This fault should be considered with close attention, because a single unsatisfiable dependency can bring the whole system to a halt/deadlock (see also discussion of business process deadlocks in Section 3.2.6). Wherever possible, the platform should provide means for statically checking circular dependencies in the event processing business logic.

3.2.5 Faults in Sensor Networks

Wireless sensor networks (WSNs) consist of a collection of densely deployed sensor nodes whose purpose is to sense and process information from the environment [2]. The key challenges intrinsic to WSNs arise from the circumstance that sensor nodes are limited in power, often prone to failures, and connected by unreliable communication channels. Moreover, the position of sensor nodes is not static and the topology of a WSN changes frequently. An adaptive and fault-tolerant routing mechanism is therefore required.

Figure 6 contains two fault cases that are related to **Unreachability**. Firstly, unreachability on the **Network** level means that a sink is unable to receive any messages from a source node, either because its communication link is down, the signal is noisy, or the node is out of range of any other nodes and hence there is no physical path from source to sink. If there is a possible path between two nodes, but a packet (or message) does not find its way to the receiver, we speak of a **Routing** related unreachability, also denoted path fault [67]. For instance, a possible reason may be that the routing algorithm partitions a set of network nodes into multiple routing domains, which are connected by a single coordinator. If this coordinator fails to work properly, the sub-networks become disjoint and messages from one domain cannot reach another domain.

Since positions of nodes and topologies in WSN can change frequently, tasks and node responsibilities are often assigned dynamically. If a task is migrated from one node to another, the operator logic as well as the EPA state need to be marshalled and transmitted over the network. The duration of transmission is a critical time window because during that time it must be ensured that no events are either lost or double-processed [29]. Moreover, all dependencies to other nodes need to be updated as soon as the task has been transferred. If this complex procedure does not complete transactionally safe, we speak of **Migration Failure**.

Hardware constraints play an important role in WSNs. Due to unreliable transmitters or noisy signals, it is possible that **Transmission Bit Flips** occur. A bit flip simply means that part of the data has changed its representation (e.g., from "0" to "1") during transmission and that neither side of the communication realized this error. Various error-detection codes (like checksums or parity bits) have been proposed to avoid transmission errors in WSNs, and there is an obvious tradeoff between degree of error-robustness and computational complexity [2].

Link Congestion occurs if network links operate beyond capability, induced by high amounts of data or too many senders writing to the same medium. Media access control (MAC) techniques like *Carrier Sense Multiple Access* (CSMA) regulate the access to a commonly used transmission medium, but there is a practical limitation to the number of devices served by the same network [2]. A link congestion is different from an overloaded channel, because network links may be shared with external systems, whereas channels are considered an internal platform component.

3.2.6 Faults in Event-Driven BPM

Event-driven business process management (EDBPM) is concerned with utilizing events to steer the orchestration of workflows and services. EDBPM can be approached from different sides: the process definition is either known explicitly and can be transformed into an EPN (e.g., [39]), or the process has an implicit model that is discovered from event logs (also denoted process mining) [69], or the challenge is to measure the fit between event logs and the process model [56]. Algorithms in these areas are complex and often make probabilistic assumptions, and hence pose a potential source of faults. A **Process Mining Fault** implicates that the (probabilistic) assumptions in the algorithm to derive a process model are inaccurate, whereas a **Transformation Fault** denotes an incorrect mapping from the original process definition to tasks on the eventing platform.

When a process or sub-process is triggered by an event, the process engine needs to be able to correlate the event to previous or currently active process instances. An **Uncorrelatable Event** occurs if the process to which the event seems to belong either does not yet exist, or does not exist anymore (because it has been finished or forcibly terminated), or has never existed (caused by faulty correlation). Note that an event which triggers an entirely new instance (and is hence not in a correlation with a previous process) does not fall into this fault category, because that reflects a regular situation. Another correlation-related defect is **Faulty Process Isolation**, which means that a set of events representing a model process instance does not correspond to reality or the actual business case. Faulty isolation is particularly problematic in process mining, because most algorithms that learn the structure of processes rely on the fact that the instances from which they learn are correct.

Process Deadlock and **Starvation** are defects related to the business logic which prevent the process from continuing its execution. In a deadlock the process arrives at a state in which the process definition allows neither termination nor moving into any successor state [68], for instance because it is waiting for a particular event. Starvation principally means that processes are competing for a resource and one process with less priority is discriminated against competitors. As an example, imagine two consumers c_1 and c_2 which are supposed to receive events of type t_1 with a fair distribution (e.g., strictly alternating order). If the event router happens to favor c_1, then c_2 may starve and wait forever or time out after not receiving an event for a while.

4. RELATED WORK

In this section we put the presented approach into perspective with existing research on models for event-based systems, fault taxonomies, and fault localization.

Fowler and Qasemizadeh [24] present a common event model for integrated sensor networks. The model distinguishes four ontologies (event, object, property and time ontology) to represent different aspects of event data. In contrast to our approach their model only focuses on the information associated with an event and does not consider processing logic or topology of EPNs. Other seminal work in the area of models and taxonomies for event-based systems, particular targeting event-based programming systems, has been published by Meier and Cahill [48].

Hadzilacos and Toueg [27] present a comprehensive study of interaction-related faults and fault-tolerance in distributed systems. They discuss various concepts of reliable message delivery, with core focus on message broadcasts. Based on a formal framework, the authors discuss issues such as timeliness or correct ordering of messages. Their work is highly influential for event-based systems, particularly for the fault types concerning event channels in our proposed model.

Westermann and Jain [73] study commonalities in event-based multimedia applications and discuss features that a common event model should contain. Although the features are largely tailored to multimedia, some aspects apply to event-based systems in general, such as *common base representation*, *application integration*, *common event management infrastructure*, and *common event exploration and visualization tools.* We extend their ideas and argue that integration of different views on event-based systems is vital to improve system dependability. Our model does not yet capture some aspects proposed in [73], most notably uncertainty support, and experiential aspects, which are described as *"ways of exploring and experiencing a course of events to let them [the users] gain insights into how the events evolved".*

Fault taxonomies for service-oriented architecture (SOA) have been discussed in [11] and [14]. The taxonomies are tailored to issues related to service-based computing and Web services (e.g., service discovery, binding, or composition), whereas we focus on specifics of event-based systems.

The authors of [31] have recently proposed a fault injection framework for assessing partial fault tolerance (PFT) of stream processing applications. Their work is tailored to high-frequency data streams and bursty tuple loss. In PFT, there is not only a notion of absolute faults but also of output quality degradation. The level of quality loss is measured using an application-specific output score function. In future work, we also strive to extend our model and approach to support PFT and output quality metrics.

Other researchers' studies focus on the development of software and investigate faults primarily on the source code level. In [19] three main types of faults are identified: *missing*, *wrong* and *extraneous* code constructs. The classification was applied to *diff* and *patch* files of open source projects. The metrics indicate that simple programmer mistakes account for a large portion of faults. For general mistakes like faulty synchronization their approach is certainly applicable in event processing platforms, but with complex interactions in place the relation between failure and responsible artifact becomes hard to assess. A common fault model as presented here can greatly simplify this search.

The work of Steinder and Sethi [63] surveys approaches and techniques for fault localization in computer networks, largely focusing on graph-theoretic fault propagation models like dependency networks and causality graphs. Their contribution provides much more general granularity than our work and parts of the faults discussed here are covered by their approach (e.g., detecting circular dependencies in EPNs). While general fault models have their justification, we argue that it is the fine granularity and domain-specific knowledge that adds to the strength of our approach.

5. CONCLUSION

So far, the emerging research field of distributed event-based systems has not yet come to a common and unified understanding of faults. In this paper we take a step ahead in this direction and present a unified fault taxonomy based on a common model for event-based systems. The taxonomy provides dimensions to obtain a comprehensive allround picture of the system artifacts as well as potential manifestations and sources of faults. We discuss 30 fault instances that cover all fault types and elements of our common model.

Profound research on dependability and fault-tolerance has been conducted on specific topics in different sub-areas of event processing, and we argue that combining these efforts is a potential leap forward on the pathway towards engineering dependable event-based systems. The established unified model and fault taxonomy open a variety of exciting future research directions.

First and foremost, we are working on a prototype which implements and applies the presented model to real systems, with the aim of fault diagnosis [30] and fault injection [31, 32]. The core idea is to provide well-defined interfaces to keep the model in sync with the real target system. In one direction, from the real system to the model, monitoring techniques will be employed to update the model in case the system changes (e.g., if a new EPA is instantiated or new events are buffered). In case of any faulty behavior we expect to be able to assist in the systematic fault diagnosis, using the domain knowledge and the data gathered from monitoring. Reversely, changes in the model will be reflected in the real system, using specific interfaces that need to be implemented by the target system. This way it will be possible to inject targeted faults into the platform (e.g., event loss or node failure) and to analyze how the real system reacts to these faults.

Our second goal for future work is to combine monitoring mechanisms and machine learning techniques to monitor the system model of real event processing platforms, in order to derive complex model artifacts, processing logic and event provenance from monitoring primitives. This will allow for concretization of dynamic and loosely structured EPNs which lack proper documentation and self-reflection.

Moreover, we plan to develop a community-driven Web platform that enables collaborative definition of extensible system models and fault taxonomies. Finally, we envision that the technology-agnostic system model will facilitate automatic migration of eventing business logic between platforms of different vendors.

6. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 257483 (Indenica).

7. REFERENCES

[1] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *2nd CIDR*, 2005.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, Elsevier, 2002.

[3] T. Aslam, I. Krsul, and E. Spafford. Use of a taxonomy of security faults. *19th NISSC*, 1996.

[4] A. Avizienis and Y. He. Microprocessor entomology: a taxonomy of design faults in COTS microprocessors. In *Dependable Comput. for Crit. Applications 7*, 1999.

[5] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE TDSC*, 1:11–33, 2004.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *21st PODS*, pages 1–16, 2002.

[7] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30:109–120, 2001.

[8] G. Banavar, T. Chandra, R. Strom, and D. Sturman. A case for message oriented middleware. In *SDC*, 1999.

[9] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *3rd CIDR*, pages 363–374, 2007.

[10] R. V. Binder. Testing object-oriented software: a survey. *STVR Journal*, 6(3-4):125–252, Wiley, 1996.

[11] S. Bruning, S. Weissleder, and M. Malek. A Fault Taxonomy for Service-Oriented Architecture. In *10th IEEE HASE*, pages 367–368, 2007.

[12] Z. Butler and D. Rus. Event-based motion control for mobile-sensor networks. *Pervasive Comp.*, 2(4), 2003.

[13] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3), 2001.

[14] K. S. Chan, J. Bishop, J. Steyn, L. Baresi, and S. Guinea. A fault taxonomy for web service composition. In *7th ICSOC - Workshops*, 2009.

[15] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[16] C. Ciordas, T. Basten, A. Radulescu, K. Goossens, and J. Meerbergen. An event-based network-on-chip monitoring service. In *9th IEEE HLDVT*, 2004.

[17] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *10th ACM SIGOPS (workshops)*, 2002.

[18] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, 2003.

[19] J. Duraes and H. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE TSE*, 32(11), 2006.

[20] E. Elnozahy, L. Alvisi, Y.-M. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comp. Surveys*, 34(3), 2002.

[21] O. Etzion and P. Niblett. *Event Processing in Action.* Manning Publications Co., 2010.

[22] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35:114–131, 2003.

[23] L. Fiege, F. Gartner, O. Kasten, and A. Zeidler. Supporting Mobility in Content-Based Publish/Subscribe Middleware. In *Middleware*, 2003.

[24] C. Fowler and B. Qasemizadeh. Towards a Common Event Model for an Integrated Sensor Information System. In *Workshop on the Semantic Sensor Web*, 2009.

[25] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *4th VDM Europe Symp. on Formal Softw. Dev.*, pages 31–44, 1991.

[26] D. Gelernter. Multiple tuple spaces in linda. *Parallel Architectures and Languages Europe*, 366:20–27, 1989.

[27] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

[28] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGPLAN Notices*, 35:93–104, 2000.

[29] W. Hummer, P. Leitner, B. Satzger, and S. Dustdar. Dynamic migration of processing elements for optimized query execution in event-based systems. In *OnTheMove Federated Conferences*, 2011.

[30] R. Isermann. Model-based fault-detection and diagnosis - status and applications. *Annual Reviews in Control*, 29(1):71–85, 2005.

[31] G. Jacques-Silva, B. Gedik, H. Andrade, K.-L. Wu,

and R. Iyer. Fault injection-based assessment of partial fault tolerance in stream processing applications. In *5th DEBS*, 2011.

[32] P. Jalote. *Fault tolerance in distributed systems.* Prentice Hall, 1994.

[33] Z. Jerzak and C. Fetzer. Bloom filter based routing for content-based publish/subscribe. In *2nd DEBS*, 2008.

[34] L. Krishnamachari, D. Estrin, and S. Wicker. The impact of data aggregation in wireless sensor networks. In *22nd DCS - Workshops*, pages 575–578, 2002.

[35] C. Krügel, T. Toth, and C. Kerer. Decentralized event correlation for intrusion detection. In *4th ICISC*, 2002.

[36] G. T. Lakshmanan, Y. G. Rabinovich, and O. Etzion. A stratified approach for supporting high throughput event processing applications. In *3rd DEBS*, 2009.

[37] J. Laprie, editor. *Dependability: Basic Concepts and Terminology.* Springer, 1992.

[38] M. Leszak, D. E. Perry, and D. Stoll. A case study in root cause defect analysis. In *22nd ICSE*, 2000.

[39] G. Li, V. Muthusamy, and H.-A. Jacobsen. A distributed service-oriented architecture for business process execution. *ACM TWEB*, 4:2:1–2:33, 2010.

[40] C. Liebig, M. Cilia, and A. Buchmann. Event composition in time-dependent distributed systems. In *4th CoopIS*, pages 70–78, 1999.

[41] D. Luckham et al. Specification and analysis of system architecture using Rapide. *IEEE TSE*, 21(4), 1995.

[42] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman, 2001.

[43] D. C. Luckham and B. Frasca. Complex Event Processing in Distributed Systems. *Analysis*, 28, 1998.

[44] S. Mahambre, M. Kumar, and U. Bellur. A Taxonomy of QoS-Aware, Adaptive Event-Dissemination Middleware. *IEEE Internet Comp.*, 11(4):35–44, 2007.

[45] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE TSE*, 31, 2005.

[46] M. Mansouri-Samani and M. Sloman. GEM: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2), 1997.

[47] D. McCarthy and U. Dayal. The architecture of an active database management system. In *SIGMOD'89.*

[48] R. Meier and V. Cahill. Taxonomy of Distributed Event-Based Programming Systems. *Computer Journal*, 48(5):602–626, 2005.

[49] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. Advanced event processing and notifications in service runtime environments. In *DEBS*, 2008.

[50] C. Moxey et al. A conceptual model for event processing systems. *IBM Redguide publication*, 2010. http://www.redbooks.ibm.com/abstracts/redp4642.html.

[51] G. Mühl, L. Fiege, and P. R. Pietzuch. *Distributed event-based systems.* Springer, 2006.

[52] E. Nakamura, A. Loureiro, and A. Frery. Information fusion for wireless sensor networks: Methods, models, and classifications. *ACM Computing Surveys*, 39, 2007.

[53] P. Pietzuch and J. Bacon. Hermes: a distributed event-based middleware architecture. In *ICDCS*, 2002.

[54] P. A. Porras and P. G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. *20th NISSC*, pages 353–365, 1997.

[55] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1), 1989.

[56] A. Rozinat and W. van der Aalst. Conformance testing: Measuring the fit and appropriateness of event logs and process models. In *BPM Workshops*, 2006.

[57] S. Rozsnyai, A. Slominski, and G. T. Lakshmanan. Discovering event correlation rules for semi-structured business processes. In *5th DEBS*, 2011.

[58] L. Ruiz, I. Siqueira, L. Oliveira, H. Wong, J. Nogueira, and A. Loureiro. Fault management in event-driven wireless sensor networks. In *7th ACM MSWiM*, 2004.

[59] A.-W. Scheer, O. Thomas, and O. Adam. *Process-Aware Information Systems*, chapter Process Modeling using Event-Driven Process Chains. Wiley, 2005.

[60] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *3rd DEBS*, 2009.

[61] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *ACM SIGMOD Conference*, pages 827–838, 2004.

[62] G. Sharon and O. Etzion. Event-processing network model and implementation. *IBM Systems Journal*, 47(2):321–334, 2008.

[63] M. Steinder and A. S. Sethi. A survey of fault localization techniques in computer networks. *Science of Computer Programming*, 53(2), Elsevier, 2004.

[64] R. Stephens. A survey of stream processing. *Acta Informatica*, 34:491–541, 1997.

[65] R. N. Taylor et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE TSE*, 22(6):390–406, 1996.

[66] J.-Y. Tigli et al. WComp middleware for ubiquitous computing: Aspects and composite event-based Web services. *Annales des Télécommunications*, 64, 2009.

[67] S. Tilak, N. B. Abu-Ghazaleh, and W. Heinzelman. A taxonomy of wireless micro-sensor network models. *ACM SIGMOBILE MC2R*, 6(2):28–36, 2002.

[68] W. van der Aalst. Formalization and verification of event-driven process chains. *IST*, 41, Elsevier, 1999.

[69] W. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: discovering process models from event logs. *IEEE TKDE*, 16(9):1128–1142, 2004.

[70] A. v.d. Goor and Z. Al-Ars. Functional memory faults: a formal notation and a taxonomy. In *VLSI Test*, 2000.

[71] R. von Ammon et al. Existing and future standards for event-driven business process management. In *3rd DEBS*, 2009.

[72] F. Wang, S. Liu, P. Liu, and Y. Bai. Bridging Physical and Virtual Worlds: Complex Event Processing for RFID Data Streams. In *10th EDBT*, 2006.

[73] U. Westermann and R. Jain. Toward a common event model for multimedia applications. *MultiMedia*, 2007.

[74] M. Wieland, D. Martin, O. Kopp, and F. Leymann. SOEDA: A Methodology for Specification and Implementation of Applications on a Service-Oriented Event-Driven Architecture. In *12th BIS*, 2009.

[75] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.