Testing Event-driven Applications with Automatically Generated Events

Maximilian Völker, Sankalita Mandal, and Marcin Hewelt

Hasso Plattner Institute at the University of Potsdam, Germany
{firstname.lastname}@hpi.de

Abstract. To make use of the abundance of events produced by smart devices, social media, or the internet-of-things (IoT), implemented business processes need to integrate external events. Thus, processes are event-driven applications that need to be tested correspondingly. However, testing with real-world events is hard, because those events cannot be controlled. Therefore, we suggest to use generated events for testing and implement an event generator component and demonstrate our approach using an use case from the IoT domain. This demo is aimed at practitioners and academics working with event-driven applications.

1 Introduction

In today's interconnected world more and more events are produced by smart devices, sensors, in social media, via APIs, and IoT, that companies need to take into account [1]. Event-driven application development is a paradigm of software development, which bases its application logic on reacting to received events [2]. Implemented business processes that are aware of and connected to external event sources can be viewed as event-driven applications, because cases are started when events occur, activities are aborted and so on.

Testing is an essential part of software development ensuring that the software operates as expected and no bugs are introduced during coding (regression testing). In case of business processes running in a process engine we need to make sure that new cases are started correctly, event-based gateways and intermediate events work as expected, and that exception handling through boundary events functions. When high-level events are employed [3], the aggregation of event patterns needs to be tested as well.

However, accessing real-world events for testing is problematic for several reasons: a) the time when they occur and their content is hard to control, b) exceptional events occur seldom or not at all during the test phase, c) sometimes there is not enough event data available, and d) it can be costly to connect to some event sources. To mitigate this problem of testability of event-driven business processes, we suggest to automatically generate events, thus providing a mock event stream for testing. Therefore, we extended our event processing platform Unicorn[1] with *Event Generator*, which is configurable and generates events with random attribute values taken from various ranges and distributions.

[1] https://bpt.hpi.uni-potsdam.de/UNICORN

We demonstrate our testing approach using a use case from the IoT domain. The use case is implemented as a business process running in the Chimera engine[2], which is connected to Unicorn, listening for events.

2 Events in Business Processes

Background. Business process management is the key for organizations to model, execute, monitor and continuously improve their processes [4]. A business process is a sequence of activities collectively realizing a specific business goal. One process can be used as the blueprint for several process instances. The de-facto standard for modeling the processes is Business Process Model and Notation (BPMN 2.0) [5]. Besides activities and gateways, a process model contains constructs to model events, which are either consumed (catching events) or produced (throwing events) during process execution. Catching events are essentially the environmental occurrences relevant for a process [6]. Depending on its usage, a catching event can trigger a process (start event), initiate an exceptional path (boundary event) or determine the branch to execute (event-based gateway).

The catching events needed for process execution can be produced by other processes or they can be generated from external sources [7]. Events can be atomic or simple which does not take any time to occur, e.g., a sensor update. On the other hand, complex events are the higher-level events produced by aggregating the simple events, e.g., a traffic congestion which is lying ahead of a truck. Using complex event processing (CEP) techniques different operations can be performed on event streams from single or multiple sources [3]. A CEP platform is able to connect to multiple event sources, parse the events generated in different formats and aggregate them based on some pre-defined rules [8]. Event consumers such as process engines can subscribe to a CEP platform for certain events. Once matching events occur, the subscribers are notified about it.

Use Case Description. The use case shown in Figure 1 is taken from our project partner who are involved in producing smart heating systems for home usage.

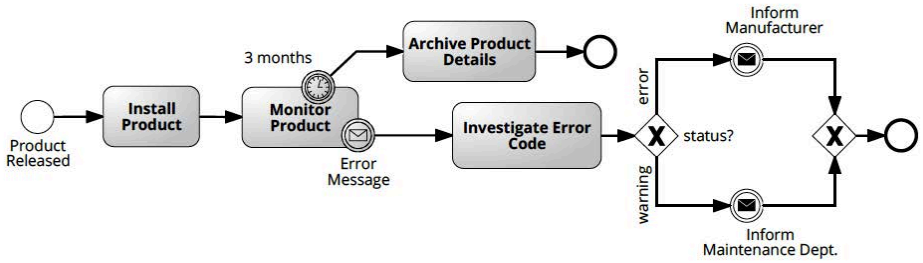

Fig. 1. Motivational example

After the heating system is installed, it is monitored for three months. If there is no error report in these three months then the product is considered

[2] `https://bptlab.github.io/chimera`

as tested in real scenario and product details are archived. In case there is an error message, the error code is investigated and corresponding departments are contacted to take action. The timer event and error event are shown as BPMN boundary events. At occurrence, each of them interrupts the activity `Monitor Product` and leads the process execution to the following exceptional path.

Now, there can be two kinds of error reports, namely, *Warning* and *Error* and either the maintenance department or the manufacturer is contacted respectively as reaction to the event. Under the hood, the CEP platform collects the sensor data sent by the heating systems and the complaints reported by the customers in the product feedback website and aggregates them to produce the error reports. For example, one rule for generating a *Warning* can be to receive more than three technical failures within a week, whereas an *Error* might be produced if the actual temperature differs from the temperature setting specified. Since we do not expect the error reports to occur in real life but still want to make sure that in case it happens the needed action is taken, we need to simulate the situation.

3 Event Generator

The event generator is implemented as an extension of the event-processing platform Unicorn. As Unicorn manages the structure of event types it processes, the event generator can use the existing event information from Unicorn and enrich the real-world event stream with artificial events.

Architecture. The event generator in Unicorn is split into three components: a handler for user input, a manager for attribute dependencies within event types and the core logic of the event generator. The user input handler has data type specific implementations, like String or Integer. It defines available computation methods, validates input and generates attribute values based on user's input and the chosen method (see *Features* below). The dependency manager component is responsible for managing dependencies between two attributes of an event type, e.g. the error description depends on the error code. The event generator core connects the other two components.

For generating artificial testing events, the user first selects the event type and number of events to generate. For each attribute of the selected event type possible values can be specified to choose from during generation. Behind each attribute an user input handler is defined. After submitting the form, these are passed to the event generator. The event generator retrieves the values computed by the handlers and combines them into new events, complying with the dependencies configured for the event type. After a set of artificial events is created, they are passed to the event replayer, an existing component of Unicorn. Considering the events' timestamps, it inserts the events into the event stream processed by Unicorn. Therefore the artificial events do not need to be inserted all at once but can be replayed like a real stream of events.

Features. The current implementation allows to specify value ranges and computation methods for each attribute separately (see Figure 2). Following the implementation of Unicorn, the data types String, Integer, Float and Date are supported and currently, values for all of them can be generated using uniform distribution. Additionally, normally distributed Float and Integer attribute values can be produced. To provide a fast generation of test events, default values are defined for each data type. For more realistic events, dependencies between attributes can be configured. They define that the value of one attribute depends on the value of another attribute, in our use case for example, the id of an error can be connected to a corresponding error message. For example, error id 102 is mapped to error message "No water", while error ids 103-110 are mapped to "generic error". When generating events, the dependencies are evaluated and when a mapping rule matches the value of the dependent attribute is set accordingly. Finally, the event generator allows to export and import its settings, which eases reuse and sharing, especially when working with many attributes and dependencies or in a team.

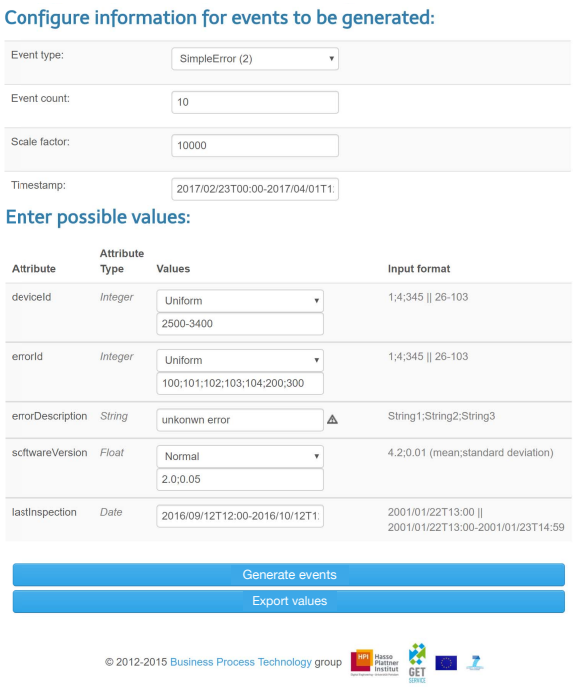

Fig. 2. Event generator interface

Discussion. The presented implementation is stable, tested and provides basic features required for testing event-driven processes. We plan to further extend the event generator to support various types of distributions, especially user-defined distributions and configurable default values for event types. At the moment the default values are defined globally independent of the event type. This might be sufficient to generate events with a given structure but the default values are seldom meaningful. Therefore configurable default values for each event type will open the possibility to generate events fast without user input and with events containing common values for the domain. To test or simulate certain scenarios a normal or uniform distribution of values might not be enough, e.g., if 80% of the events need to have normal values and 20% should be faulty, then this is only possible if the user can define the probability of values manually.

To use the event generator for test cases, the event set should not differ for each execution. Here, the generator can be combined with the event replayer again: by saving the event generator's output, the same set of events can be reused as needed by passing it to the event replayer. Now the tester can be certain that the occurred events are the same as the last test execution and the test result only depends on the code.

The generated events are then aggregated based on pre-defined rules to create higher-level events. The Chimera engine gets notification about these higher-level or business events and reacts on them as prescribed in the model [9].

4 Conclusion and Future Work

The work focuses on the lack of events to test event-driven applications and develops an event generator to address the problem. The event generator component together with the event replayer increases the testability of event-driven applications, and especially business processes relying on external events.

However, event generation and replay is only a part of a testing framework for event-driven business processes. Additionally, service calls, manual tasks, user input and decisions, among others need to be tested. Also, as the generated events are decoupled from their original event sources, it is necessary to test the correct receiving and decoding of events from these sources separately. An integrated testing framework for event-driven processes is left for future work. Furthermore, we aim to make generated events more realistic by analyzing historical event data and learning their distributions.

The event generator is a mature and tested component of the Unicorn event processing platform, available since release 1.6. The screencast is available at `https://bpt.hpi.uni-potsdam.de/UNICORN/EventGenerator`.

References

1. Greengard, S.: The internet of things. MIT Press (2015)
2. Hinze, A., Buchmann, A.P.: Principles and applications of distributed event-based systems. Hershey, PA : Information Science Reference (2010)
3. Barros, A., Decker, G., Grosskopf, A.: Complex events in business processes. In: Business Information Systems, Springer (2007)
4. Weske, M.: Business Process Management - Concepts, Languages, Architectures, 2nd Edition. Springer (2012)
5. OMG: Business Process Model and Notation (BPMN), Version 2.0. `http://www.omg.org/spec/BPMN/2.0/` (January 2011)
6. Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications (2010)
7. Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley (2010)
8. Herzberg, N., Meyer, A., Weske, M.: An event processing platform for business process management. In: EDOC, IEEE (2013)
9. Beyer, J., Kuhn, P., Hewelt, M., Mandal, S., Weske, M.: Unicorn meets Chimera: Integrating External Events into Case Management. In: BPM Demo Session, CEUR-WS.org (2016)