Change Patterns for Supporting the Evolution of Event-Based Systems

Simon Tragatschnig, Huy Tran, and Uwe Zdun

Research Group Software Architecture
University of Vienna, Austria
{simon.tragatschnig,huy.tran,uwe.zdun}@univie.ac.at

Abstract. As event-driven architectures consist of highly decoupled components, they are a promising solution for facilitating high flexibility, scalability, and concurrency of distributed systems. However, the evolution of an event-based system is often challenging due to the intrinsic loose coupling of its components. This problem occurs, on the one hand, because of the absence of explicit information on the dependencies among the constituting components. On the other hand, assisting techniques for investigating and understanding the implications of changes are missing, hindering the implementation and maintenance of the changes in event-based architectures. Our approach presented in this paper aims at overcoming these challenges by introducing primitive change actions and higher-level change patterns, formalized using trace semantics, for representing the modification actions performed when evolving an event-based system. Our proof-of-concept implementation and quantitative evaluations show that our approach is applicable for realistic application scenarios.

1 Introduction

Event-driven architectures are a promising solution for developing distributed systems that facilitates high flexibility, scalability, and concurrency [6, 8]. An event-based system consists of a number of computational or data components that communicate with each other by emitting and receiving events [8]. Each component may independently perform a particular task, for instance, accessing a database, checking a credit card, interacting with users, or writing to a log file. The execution of a component can be triggered by some particular events, which are called the *input events*. In turn, a component can also emit one or many *output events*. The transfer of events among the components is performed through an event channel. Therefore, every component is totally unaware of the others. This way, the event-based communication style can support a high degree of flexibility. For instance, it enables replacing or altering any component (e.g., with a bug-fixed or upgraded version) or to change the execution order of the components (e.g., re-routing, skipping, or adding some components) whilst the system is running. There is a rich body of work in different research areas that investigate and exploit the prominent advantages of event-based communication styles such as middleware infrastructure [4], event-based coordination [2], active database systems [10], and service-oriented architectures [9], to name but a few.

R. Meersman et al. (Eds.): OTM 2013, LNCS 8185, pp. 283–290, 2013.

Unfortunately, by introducing additional degrees of flexibility, the loose coupling in event-based systems also increases the difficulty and uncertainty in maintaining and evolving these systems. Implementing specific changes in an event-based system is challenging, because of the absence of explicit dependencies among constituent components makes understanding and analyzing the overall system composition difficult.

We present in this paper a novel approach for supporting the evolution of event-based systems. In particular, we introduce fundamental abstractions for describing primitive modifications that can be used to alter an event-based system. These primitive actions capture the low-level actions for modifying event-based systems, and therefore, can be efficiently leveraged by technical experts. On top of these primitive actions, we devise a number of high-level abstractions described as *change patterns* for event-based systems along with their formal descriptions based on trace semantics. These patterns encapsulate essential change actions that recur in many software systems. A certain evolution requirement can be implemented through low-level abstractions by applying an individual or a chain of primitive actions or at higher level of abstractions by using one or more change patterns, respectively. Our quantitative evaluations show the applicability of our approach for realistic application scenarios.

The paper is structured as following. In Section 2, we introduce some preliminary concepts and definitions in the context of event-based software systems and the trace semantics used to formalize our change patterns. Section 3 describes the fundamental concepts and abstractions of our approach for supporting the evolution of event-based systems. Section 4 presents evaluations of its productivity, based on our proof-of-concept implementation. The related literature is discussed in Section 5. We summarize the main contributions and discuss the planned future work in Section 6.

2 Preliminaries

Our approach aims at introducing techniques for implementing changes in event-based systems. Without loss of generality, we adopt the notion that a generic event-based system comprises a number of components performing computational or data tasks and communicating by exchanging events through event channels [8]. The inherent loosely coupled nature of the participating components of an event-based system makes it challenging to understand and implement changes. This issue is, on the one hand, due to the absence of explicit dependency information. On the other hand, software engineers have to deal with the complexity and every technical details of the underlying event-based systems because of the lack of appropriate abstractions for handling and managing changes. To aid the software engineers in better analyzing and applying changes for an event-based system, we make some basic assumptions that slightly reduce the system's non-determinism while still preserving a large degree of the flexibility and adaptability: (1) each component exposes an event-based interface that specifies a set of events that the component expects (aka the *input events*) and a set of events that the component will emit (aka *the output events*); (2) the execution of a component is triggered by its input events; and (3) after a component is executed, it will eventually emit its output events.

These requirements aim at specifying the semantics of the typical behavior of an event-based component and making some pragmatic assumptions that are slightly constraining the non-deterministic nature of event-based systems. Please note that the first requirement does not forbid the alteration of a component's input and output events but only enables us to be able to observe the input/output event information at a certain point in time. The major advantage of this perspective on event-based systems is that it supports extracting dependency information at any time without requiring access to the (currently deployed) source code. In addition, this requirement is pragmatic in case third-party components are used as they are often provided as black-boxes with documented interfaces.

Please also note that these requirements can be satisfied for most event-based components without change or with reasonable extra costs (e.g., for developing simple wrappers in case of using third-party libraries and components). Most of the existing event-based systems already support equivalent concepts [8]. For demonstration purpose, we leverage the Dynamic Event-driven Actor Runtime Architecture (DERA) framework [16] that provides basic concepts for modeling and developing event-based systems and supports the three requirements. The DERA concepts can easily be generalized to the concepts found in other event-based systems.

In DERA, a computational or data handling component is represented by an *event actor* (or *actor* for short). An *event* can be considered essentially as "any happening of interest that can be observed from within a computer" [8] (or a software system). An event might contain some attributes such as its unique identification, timing, data references, and so on [8]. DERA uses the notion of *event types* to represent a class of events that share a common set of attributes. To encapsulate a logical group of related actors (for instance, actors that perform the functionality of a certain department or organization), DERA provides the concept of *execution domains*. Two execution domains can be connected via a special kind of actor, namely, *event bridge*, which receives and forwards events from one domain to the other [16]. Well-defined actor interfaces support us in analyzing and performing runtime changes in event-based systems, such as substituting an event actor by another with a compatible port or changing the execution order of event actors by substituting an actor with another [16].

In this paper, we leverage trace semantics [3] for describing the observed behavior of event-based systems and the semantics of the proposed change patterns. One of the major advantages of trace semantics, which is very suitable for our approach, is that the underlying system can be treated as a black box and its behavior is described in terms of the states and actions that we observe from outside.

3 Approach

The implementation of a particular change in an event-based system involves defining the relevant actions (e.g., adding or removing components, enabling or disabling components, altering the components' inputs or outputs, or adjusting the execution order of components) and carrying out these actions while taking into account the consequences (as other components might be affected by these actions). That is, in order to enact a change in an event-based system, the software engineers have to deal with many technical details at different levels of abstraction, which is very tedious and error-prone.

Weber et al. identify a set of change patterns that recur in many of existing software systems [17]. These patterns are specific for process-aware information systems (PAIS) where the execution of the software system is bound to a process schema. Changes mostly can not be done during runtime [1, 13, 15]. These patterns are specific for process-aware information systems (PAIS) where the execution of the software system is bound to a process schema, a prescribed rigid description of the behavior flow, and therefore, mostly can not be changed during runtime or just slightly deviated from the initial schema [1, 13, 15]. As a result, these approaches are not readily applicable for event-based systems where components are highly decoupled and the dependencies between components are subject to change at any time, even during the execution of the systems. Nevertheless, the aforementioned patterns provide a basis for describing changes of the behavior in any information systems.

In our work, we investigate and adapt these patterns in the context of event-based software systems that are different to PAISs because there are no prescribed execution descriptions and the constituent elements of a system and their relationships can be arbitrarily changed at any time. In order to deal with the complexity and the large degree of flexibility of event-based systems, we aim at supporting system evolution at different levels of abstraction. We introduce low-level primitives for encapsulating the basic change actions, such as adding or removing an event or an actor, replacing an event or actor, and so forth.

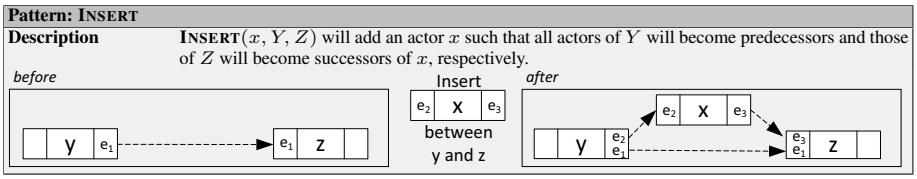
The definitions of these primitives are given in Table 1. They describe simple, primitive low-level actions for populating and modifying event-based systems that conform to the definitions we provided in Section 2. Based on these primitives, in the following we present change patterns for event-based systems with which the software engineers can easier describe and apply desired changes at a higher level of abstraction.

On top of the aforementioned primitives, we introduce change patterns for event-based systems. These patterns extend the change patterns that are frequently occurring and supported in most of today's information systems according to the survey of Weber et al. [17]. In this section, we present the change patterns along with their formal descriptions and abbreviated proofs for correctness. We describe these pattern based on the widely accepted intention of the developers as observed and documented in [17] and discuss potential variants and extensions of these pattern.

Due to space limitations, we opt to present the **INSERT** in detail. We also realized a set of other patterns in the same way as **INSERT**. An overview of our patterns as well as their evaluation can be found in Table 2.

Pattern INSERT. As an event-based system evolves, additional functionality is often added. We use a simple but realistic example of an online shopping system to illustrate the situation. In this system, to complete an order, the customer can use two addresses: one for billing and one for shipping. Let us consider the following scenario: In the first system deployment, the shipping address is also used for the billing address. During evolution of the system, a new component for adding an additional billing address should get inserted. Normally, in event-based systems the developers would have to deal with every technical details of implementing new components and exchanging events among the components. The **INSERT** pattern aims at encapsulating and hiding such details to help the developers to focus on defining the behaviors of the new components

and specifying the desired inputs and outputs of the actors. The example visualization of the pattern below shows the change pattern to be applied in the middle. On the left side, the event-based system before the change is depicted, and on the right side it is depicted after the change.



An event-based system $\mathcal{S}$ is represented in DERA by a 2-tuple $(\mathcal{A}, \mathcal{E})$, where $\mathcal{A}$ is the set of event actors and $\mathcal{E}$ is the set of event types exchanged by these actors. Let $\bullet x$ (resp. $x\bullet$) be the set of input (resp. output) events of an actor x. The **INSERT** pattern (represented by the function p) that transforms an execution domain $\mathcal{S}(\mathcal{A}, \mathcal{E})$ into $\mathcal{S}'(\mathcal{A}', \mathcal{E}')$, i.e., $p : \mathcal{S} \xrightarrow{\text{INSERT}(x,Y,Z)} \mathcal{S}'$, can be defined as follows.

$$\begin{aligned}
&\mathcal{A}' = p(\mathcal{A}) = \mathcal{A} \cup x \\
&\mathcal{E}' = p(\mathcal{E}) = \mathcal{E} \cup x\bullet \cup \bullet x \\
&Y' = p(Y) : \forall y \in Y : y'\bullet = y\bullet \cup \bullet x, \text{where } y' = p(y) \\
&Z' = p(Z) : \forall z \in Y : \bullet z' = \bullet z \cup x\bullet, \text{where } z' = p(z)
\end{aligned} \tag{1}$$

The developers may want to use a variant of the **INSERT** pattern in which the transitions from the actors of Y to those of Z will be strictly redirected through x, i.e., $y' \rightarrow x' \rightarrow z'$. A formal description of the variant can be adapted from Equation (1) as:

$$\begin{aligned}
&\bullet y' = \bullet y \setminus \{e | e \in x\bullet \cap \bullet y \ \wedge \ e \notin a\bullet, \forall a \in \mathcal{A}\}, \text{where } y' = p(y) \\
&y'\bullet = y\bullet \setminus \{e | e \in y\bullet \cap \bullet z \ \wedge \ e \notin a\bullet, \forall a \in \mathcal{A}\}, \text{where } y' = p(y) \\
&\bullet x' = \bullet x \cup y'\bullet, \text{where } x' = p(x), y' = p(y) \\
&\bullet z' = x\bullet \cup \bullet z' \setminus \{e | e \in y\bullet \cap \bullet z \ \wedge \ e \notin a\bullet, \forall a \in \mathcal{A}\} \text{where } z' = p(z)
\end{aligned} \tag{2}$$

The specification of $\bullet y'$ in Equation (2) is to adjust any transition $x \rightarrow y$ that exists before changing. We alter the output of y', i.e., $y'\bullet$, and the inputs of x' and z', i.e., $\bullet x'$ and $\bullet z'$, respectively, so direct transitions $y \rightarrow z$ will be transformed to $y' \rightarrow x' \rightarrow z'$.

We devise the post-conditions for the basic case of the **INSERT** pattern according to the Equation (1) and assert that the changed system must satisfy these conditions. The conditions for the extended cases and their proofs can be achieved in the same manner.

Lemma 1. *The new state $\mathcal{S}'$ of the execution domain $\mathcal{S}$ achieved by applying the* **INSERT** *pattern, i.e., $\mathcal{S} \xrightarrow{\text{INSERT}(x,Y,Z)} \mathcal{S}'$, satisfies:*

$$\forall t \in \mathcal{T}_{S'}, \forall y \in Y : y \in t \Rightarrow y \prec x \tag{3}$$

$$\forall t \in \mathcal{T}_{S'}, \forall z \in Z : x \in t \Rightarrow x \prec z \tag{4}$$

We sketch a simple proof for Equation (3), which can be applied for Equation (4).

Proof. Let $\mathcal{S}'$ be the result of the application of the **INSERT**(x, Y, Z) pattern on $\mathcal{S}$. When an actor $y \in Y$ finishes its execution, y will emit all of its output events according to the prerequisite **R3** including the events that x is awaiting with respect to Equation 1. As a result, x will be triggered next due to **R2**. Thus, $y \prec x$. □

4 Evaluation

In the scope of our work, a proof-of-concept implementation of the primitive actions and change patterns has been developed and incorporated into the DERA framework [16]. In order to illustrate the increase of productivity using our approach, we estimate the necessary effort for manually implementing a change on an event-based system and compare these results to our change patterns. To quantify the required efforts for a change, we count the number of statements used for implementing a change. This is analogous to Line of Code metrics [5].

In Table 1, the number of statements encapsulated in the primitive change actions are presented. The first column contains a letter representing the corresponding primitive action shown in the second column. The third column, namely, E_s, depicts the number of statements needed to express the primitives. We note that only the major statements for the implementation are counted while the declarations (e.g. of packages, class body, methods or variables), comments, logging or failure handling are ignored. Also, the number of statements reflect the most simple case, handling only one predecessor and successor.

Table 1. Change Primitives for DERA-based systems

	Change Primitives	**Description**	E_s
INS	`add(Actor a)`	Add the actor a to the execution domain	9
DEL	`remove(Actor a)`	Remove the actor a from the execution domain	8
TAR	`setTarget(Actor a, ExecutionDomain d)`	Set the execution domain d for an actor a	3
PRT	`set(Actor a, Port p)`	Set a new port p for the actor a	8
DOM	`setDomain(Actor a, ExecutionDomain d)`	Set the execution domain d for actor a	2
ADD	`add(Port p, Event[] events)`	Add a set of $events$ to port p	14
REM	`remove(Port p, Event[] events)`	Remove a set of $events$ from port p	12
REPALL	`replace(Port p, Event[] events)`	Replace all events of port p with another set of $events$	7
REP	`replace(Port p, Event e1, Event e2)`	Replace event $e1$ of port p with event $e2$	11

Table 2. Change Patterns and Effort Reduction

Change Pattern	**Description**	**Primitive**	E_p	E_s	$ER\,(\%)$
INSERT	will add an actor x such that all actors of Y will become predecessors and those of Z will become successors of x, respectively	INS, 2*ADD	3	37	8.11
DELETE	will remove the actor x from the current execution domain $\mathcal{S}$	DEL	1	8	12.50
MOVE (x, y, z)	will move the actor x in a way that the actor y will become predecessor and the actor z will become successor of x, respectively	2*ADD, REPALL	3	35	8.57
REPLACE(x, y)	will substitute the actor x by the actor y	INS, DEL, 2*ADD	4	45	8.89
SWAP(x, y)	Given an actor x that precedes an actor y, this pattern will switch the execution order between x and y	4*REPALL	4	28	14.29
PARALLELIZE(x, y)	enables the concurrent execution of two actors x and y that are performed sequentially before	4*REPALL	4	28	14.29
MIGRATE$(x, \mathcal{S}_1, \mathcal{S}_2)$	will migrate an actor x from an execution domain $\mathcal{S}_1$ to another execution domain $\mathcal{S}_2$	INS, DEL, DOM, 4*ADD	7	75	9.33
Average					10.89

The result of the effort comparison between the number of statements for change primitives and the change patterns are shown in Table 2. The first column shows the name of the change pattern. The second column lists the used change primitives. The third column shows the number of change primitives E_p used by a change pattern. The third column shows the number of Java statements E_s used to implement

these change primitives. The last column shows the effort ratio ER between the change primitives and the sum of Java statements, where $ER = E_p/E_s$. The ratio shows that describing changes using our change patterns is about 11 percent of the effort compared to the code needed to implement each change manually. Using the change patterns, there are roughly 9 times (i.e., $1/11\%$) less statements needed to be written in comparison to code each change individually.

5 Related Work

Weber et al. [14, 17] identified a large set of change patterns that are frequently occurring in and supported by the most of today's process-aware information systems, where a process is described by a number of activities and a control flow is defining their execution sequence. Since the process structure is defined at design time, changing it at runtime is very difficult. Several approaches try to relax the rigid structures of process descriptions to enable a certain degree of flexibility of process execution [7, 11, 12]. Event-based systems, like DERA, provide a high flexibility for runtime changes, since only virtual relationships among actors exist. The change patterns observed by Weber et al. are designed to target PAIS in which the execution order of the elements are prescribed at design time and not changed or slightly deviated from the prescribed descriptions at runtime. Therefore, they are readily applicable for event-based systems where components are highly decoupled from each other.

6 Conclusion

Supporting the evolution of event-based systems is challenging because software engineers have to deal not only with the complexity but also a large degree of flexibility of these systems. We address this challenge in this paper by introducing novel concepts and techniques for aiding the software engineers in better analyzing and implementing particular changes on an event-based system. At the low level of abstraction, our approach provides primitive change actions that encapsulates several programming language-level and/or event exchange-level statements. Although these primitives can be used by technical experts who are able to handle technical details, it is still tedious and error prone to use them directly. We propose, at a higher level of abstraction, change patterns for event-based systems that are frequently supported and used in several information systems nowadays along with their formal descriptions. If the change patterns can be used, the effort required for a change can be significantly reduced as shown in our evaluation. A limitation of our approach is that the change patterns only perform all required changes, if the change requirements do not deviate from the specified patterns. In such cases, the changes must be manually reviewed and maybe additional changes using the primitive actions are needed. We plan to further automate changes in our future work, e.g. by supporting parameterizable variants of the change patterns and suggesting useful additional changes through tool support.

Acknowledgement. This work was partially supported by the European Union FP7 project INDENICA, Grant No. 257483 and the WWTF project CONTAINER, Grant No. ICT12-001.

References

[1] van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative workflows: Balancing between flexibility and support. Computer Science - Research and Development 23(2), 99–113 (2009)
[2] Arbab, F., Talcott, C. (eds.): COORDINATION 2002. LNCS, vol. 2315. Springer, Heidelberg (2002)
[3] Broy, M., Olderog, E.R.: Trace-Oriented Models of Concurrency. In: Bergstra, J., Ponse, A., Scott, S. (eds.) Handbook of Process Algebra, pp. 101–195. Elsevier Science B.V. (2001)
[4] Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM Trans. Comput. Syst. 19(3), 332–383 (2001)
[5] Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd edn. PWS (1998)
[6] Fiege, L., Mühl, G., Gärtner, F.C.: Modular event-based systems. The Knowledge Engineering Review 17(4), 359–388 (2002)
[7] Hallerbach, A., Bauer, T., Reichert, M.: Capturing variability in business process models: the provop approach. J. Softw. Maint. Evol. 22, 519–546 (2010)
[8] Mühl, G., Fiege, L., Pietzuch, P.: Distributed Event-Based Systems. Springer (2006)
[9] Overbeek, S., Janssen, M., Bommel, P.: Designing, formalizing, and evaluating a flexible architecture for integrated service delivery: combining event-driven and service-oriented architectures. Service Oriented Computing and Applications 6, 167–188 (2012)
[10] Paton, N.W., Díaz, O.: Active database systems. ACM Comput. Surv. 31(1), 63–103 (1999)
[11] Redding, G., Dumas, M., ter Hofstede, A., Iordachescu, A.: Modelling flexible processes with business objects. In: IEEE Conf. on Commerce and Enterprise Computing (CEC), pp. 41–48 (2009)
[12] Reichert, M., Dadam, P.: Enabling adaptive process-aware information systems with ADEPT2. In: Handbook of Research on Business Process Modeling, pp. 173–203. Information Science Reference (2009)
[13] Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies. Springer (2012)
[14] Rinderle-Ma, S., Reichert, M., Weber, B.: On the formal semantics of change patterns in process-aware information systems. In: Li, Q., Spaccapietra, S., Yu, E., Olivé, A. (eds.) ER 2008. LNCS, vol. 5231, pp. 279–293. Springer, Heidelberg (2008)
[15] Schonenberg, H., Mans, R., Russell, N.: Process flexibility: A survey of contemporary approaches. In: Dietz, L.G., Albani, A., Barjis, J. (eds.) CIAO! 2008 and EOMAS 2008. LNBIP, vol. 10, pp. 16–30. Springer, Heidelberg (2008)
[16] Tran, H., Zdun, U.: Event-driven actors for supporting flexibility and scalability in service-based integration architecture. In: Meersman, R., et al. (eds.) OTM 2012, Part I. LNCS, vol. 7565, pp. 164–181. Springer, Heidelberg (2012)
[17] Weber, B., Rinderle, S., Reichert, M.: Change patterns and change support features in process-aware information systems. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007. LNCS, vol. 4495, pp. 574–588. Springer, Heidelberg (2007)