Impact Analysis for Event-based Systems using Change Patterns

Simon Tragatschnig, Huy Tran and Uwe Zdun
Research Group Software Architecture
University of Vienna, Austria
{firstname.lastname}@univie.ac.at

ABSTRACT

Being composed of highly decoupled components, event-driven architectures are promising solutions for facilitating high flexibility, scalability, and concurrency of distributed systems. However, analyzing, maintaining, and evolving an event-based system are challenging tasks due to the intrinsic loose coupling of its components. One of the major obstacles for analyzing an event-based system is the absence of explicit information on the dependencies of its components. Furthermore, assisting techniques for analyzing the impacts of certain changes are missing, hindering the implementation the changes in event-based architectures. We presented in this paper a novel approach to supporting impact analysis based on the notion of change patterns formalized using trace semantics. A change pattern is an abstraction of the modification actions performed when evolving an event-based system. Based on this formal foundation, we introduce supporting techniques for estimating the impact and detecting undesired effects of a particular system evolution, such as dead paths, deadlocks, and livelocks. Quantitative evaluations for event-based systems with large numbers of components show that our approach is feasible and scalable for realistic application scenarios.

1. INTRODUCTION

Facilitating high flexibility, scalability, and concurrency in distributed systems is challenging. Event-driven architectures are a promising solution [13], which consist of a number of computational or data components that communicate with each other by emitting and receiving events [13]. In an event-based system, a component is totally unaware of the others and is indirectly triggered by particular events emitted by other components, which leads to a high degree of flexibility. 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 [6], event-based coordination [1], active database systems [17], and service-oriented architectures [16], to name but a few. Unfortunately, the loose coupling in event-based systems also leads to increase difficulty and uncertainty in maintaining and evolving these systems.

SAC'14 March 24-28, 2014, Gyeongju, Korea.

Understanding and implementing specific changes in event-based systems is challenging for two main reasons. Firstly, the absence of explicit dependencies among constituent components makes understanding and analyzing the overall system composition difficult. Secondly, the lack of supporting techniques for analyzing the implications of changes in an event-based system hinders the implementation and maintenance of these changes. Existing approaches for software change impact analysis are deriving dependency information from source code, assuming components are directly referenced by other components [11]. Only a few approaches are able to interpret event based architecture styles [18, 10] to derive dependency information. However, these approaches use design- or compile-time information to derive dependencies and are therefore not applicable for analyzing the impact of changes at runtime.

We proposed a novel approach for supporting the evolution of event-based systems by introducing fundamental abstractions for describing primitive modifications, namely, change primitives, that can be used to alter an event-based system [24]. More complex change patterns, for instance, substituting or moving components, can be described by composing these change primitives. In particular, our approach leverages formal descriptions of change patterns in terms of trace semantics to support change impact analysis. In order to aid the software engineers in understanding and/or adjusting particular changes, we develop algorithms for automatically determining the impact of these changes, especially the undesired or unintended effects, such as dead paths, deadlocks, and livelocks. Our quantitative evaluations show that our approach is feasible and applicable in realistic large-scale settings with considerably large numbers of constituent components.

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. The algorithms for detecting anomalies and estimating the impact of a certain change are elaborated in Section 4. Section 5 presents the evaluations of performance and scalability of our approach based on our proof-of-concept implementation as well as the estimation of the productivity gaining when applying change patterns. The related literature is discussed in Section 6. We summarize the main contributions and discuss the planned future work in Section 7.

2. PRELIMINARIES

The main focus of our work is to support change impact analysis for 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 commu-

nicating by exchanging events through event channels [13]. Due to the inherent loosely coupled nature of the participating components of an event-based system, understanding and implementing changes is challenging for software engineers. To support better applying specific changes and understand its impacts, we slightly reduced the non-determinism due to the loose coupling relationships while still preserving flexibility and adaptability by making some basic assumptions on the behavior of the constituent elements. *First*, 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*). *Second*, the execution of a component will be triggered by its input events. And *third*, a component will eventually emit its output events after its execution finishes.

Please note that the first assumption does not forbid altering a component's input and output events but only enables us to observe the input/output event information at a certain point in time. The major advantage of this premise is that it supports extracting dependency information at any time without requiring access to the source code. Indeed, this requirement is totally pragmatic in case third-party components are used as they are often provided as black-boxes with documented interfaces. In general, these requirements can be satisfied by most of existing event-based components without change or with reasonable extra costs (e.g., for developing simple wrappers to use third-party libraries and components) [13].

For demonstration purpose, we leverage the DERA framework [25] 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 many other event-based systems. In DERA, a 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" [13] (or a software system). 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, 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 [25].

DEFINITION 1 (EXECUTION DOMAIN). *A DERA execution domain $\mathcal{S}$ can be described by a 2-tuple $(\mathcal{A}, \mathcal{E})$, where $\mathcal{A}$ is the finite set of event actors that are deployed or executing within $\mathcal{S}$, and $\mathcal{E}$ is the finite set of event types exchanged by the actors of $\mathcal{S}$.*

Well-defined actor interfaces will 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 [25]. In our work, the advantage of explicitly defining actor interfaces is to enable us to capture the dependencies between the actors at a certain point in time by analyzing their inputs and outputs. The actor's interface is defined as follows.

DEFINITION 2 (ACTOR INTERFACE). *An interface $\mathcal{I}_x$ of an actor x can be described by a tuple $(\bullet x, x\bullet)$, where $\bullet x$ is a finite set of input events that x expects and $x\bullet$ is a finite set of output events that x will emit. The notions $\bullet x$ and $x\bullet$, respectively, are so-called the* input *and* output ports *of the actor x.*

To describe the observed behavior of event-based systems as well as the semantics of the proposed change patterns, we leverage trace semantics [5]. With trace semantics, 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.

DEFINITION 3 (TRACE). *Let $\mathcal{S}(\mathcal{A}, \mathcal{E})$ be an event-based system and $\mathcal{T}_\mathcal{S}$ be the set of all possible execution traces over $\mathcal{S}$. A* trace $t \in \mathcal{T}_\mathcal{S}$ *is defined as* $t = a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \ldots$, *where $a_i \in \mathcal{A}$. The notion $a_1 \rightarrow a_2$ (or $a_1 \prec a_2$ for short) denotes that a_1 precedes a_2 in the trace t.*

Given an actor $x \in \mathcal{A}$, let $\bullet x$ and $x\bullet$ be the input and output ports of x, respectively. In a trace t, a_1 precedes a_2, i.e., $a_1 \rightarrow a_2$ or more precisely as $a_1 \xrightarrow{e} a_2$, implies that there exists $e \in \mathcal{E}$ such that $e \in a_1\bullet \cap \bullet a_2$ and e causes the transition from a_1 to a_2. Hence, $a_1 \prec a_2$ also implies that $a_1\bullet \cap \bullet a_2 \neq \varnothing$.

3. CHANGE PATTERNS FOR EVENT-BASED SYSTEMS

The implementation of a particular change in an event-based system involves defining the relevant actions (e.g., adding or removing components) and carrying out these actions while taking into account the consequences (as other components might be affected by these actions). 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.

Tragatschnig et al. presented fundamental abstractions for implementing certain changes in an event-based system based on the notion of change pattern [24]. In this approach, low-level primitives are introduced for encapsulating the basic change actions, such as adding or removing an event or an actor, replacing an event or actor, and so forth. Based on these primitives, change patterns for event-based systems are defined based on the patterns that are frequently occurring and supported in most of today's information systems according to the survey presented in [26].

In this paper, we leverage the notion of change patterns and propose a formalization of the change patterns based on trace semantics in order to support impact analysis. Due to space limitation, we present a set of representative change patterns in Table1. We also discuss potential variants and extensions of the pattern. The patterns are based on the widely accepted intention of the developers as observed and documented in [26].

Change Pattern	Description
INSERT(x, Y, Z)	Add an actor x such that all actors of Y will become predecessors and those of Z will become successors of x, respectively
DELETE(x)	Remove the actor x from the current execution domain $\mathcal{S}$
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
REPLACE(x, y)	Substitute the actor x by the actor y
SWAP(x, y)	Given an actor x that precedes an actor y, this pattern will switch the execution order between x and y
PARALLELIZE(x, y)	Enable the concurrent execution of two actors x and y that are performed sequentially before
MIGRATE$(x, \mathcal{S}_1, \mathcal{S}_2)$	Migrate an actor x from an execution domain $\mathcal{S}_1$ to another execution domain $\mathcal{S}_2$

Table 1: A subset of change patterns

Formally speaking, a change pattern can be seen as a function that transforms an event-based system from one state to another. We will use the notation $p : \mathcal{S} \mapsto \mathcal{S}'$ to denote that a change pattern p is applied on an event-based system represented by a DERA execution domain $\mathcal{S}(\mathcal{A}, \mathcal{E})$.

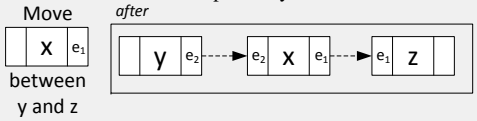
We will discuss our approach in detail through an illustrative study of the change pattern **MOVE** as this pattern conceptually embraces the functions of other patterns such as **DELETE** and **IN-**

SERT. Nevertheless, our approach can be applied in other patterns in the same manner.

Let us assume that the developers need to change the execution position of a certain component. For example, entering the shipping address was initially done before the user selects goods. Since most of the competitors ask for the shipping address after the selection of goods, the component for gathering the shipping address should be moved. This task cannot easily be done at runtime in information systems that do not use flexible communication styles such as event-based architectures because the execution order loaded into the execution engine is often based on rigid dependencies prescribed at design time. In event-based systems, on the other hand, the large degree of flexibility also leads to more complexity that the developers are confronted with. The reason is that there is a lack of adequate abstractions for supporting moving particular execution parts to different places. The **MOVE** pattern aims at overcoming this issue.

Pattern: MOVE

Description **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.



The **MOVE** pattern that transforms an execution domain $\mathcal{S}$ into $\mathcal{S}'$, i.e., $p : \mathcal{S} \xrightarrow{\textbf{MOVE}(x,y,z)} \mathcal{S}'$, is formalized as follows:

$$
\begin{array}{lcl}
\mathcal{A}' & = & p(\mathcal{A}) \equiv \mathcal{A} \\
\mathcal{E}' & = & p(\mathcal{E}) \equiv \mathcal{E} \\
\bullet y' & = & \bullet y \setminus \{e | e \in x\bullet \cap \bullet y \ \wedge \ e \notin a\bullet, \\
 & & \forall a \in \mathcal{A} \ \wedge \ a \neq x\}, \text{where } y' = p(y) \\
y'\bullet & = & y\bullet \setminus \{e | e \in y\bullet \cap \bullet z \ \wedge \ e \notin a\bullet, \\
 & & \forall a \in \mathcal{A} \ \wedge \ a \neq x\}, \text{where } y' = p(y) \\
\bullet x' & = & \bullet x \cup y'\bullet, \text{where } x' = p(x), y' = p(y) \\
\bullet z' & = & x\bullet \cup \bullet z' \setminus \{e | e \in y\bullet \cap \bullet z \ \wedge \ e \notin a\bullet, \\
 & & \forall a \in \mathcal{A}\}, \text{where } z' = p(z)
\end{array}
\tag{1}
$$

Essentially, the **MOVE** pattern alters the execution order such that the execution of x will follow the execution of y and trigger the execution of z, i.e., $y' \rightarrow x' \rightarrow z'$. According to Equation (1) we devise the post-conditions for the **MOVE** pattern, which must be satisfied by the changed system.

LEMMA 1. *The new state $\mathcal{S}'$ of the execution domain $\mathcal{S}$ achieved by applying the* **MOVE** *pattern, i.e., $\mathcal{S} \xrightarrow{\textbf{MOVE}(x,y,z)} \mathcal{S}'$, satisfies:*

$$\forall t \in \mathcal{T}_{S'}, \forall y \in Y : y \in t \Rightarrow y \prec x \tag{2}$$

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

We sketch a simple proof for Equation (2), which can be applied similarly for Equation (3).

PROOF. Let $\mathcal{S}'$ be the result of the application of the **MOVE**(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. CHANGE IMPACT ANALYSIS

It is vital for the developers who implement and deploy particular changes on a software system to understand the effect of the changes not only to the elements that are directly involved but also to the rest of the system [4]. In the previous section, we present formal descriptions and proofs of correctness of change patterns that are the foundation for understanding the direct effects of the change patterns on the involved elements. The indirect impact (aka the ripple effect) [4] of change patterns will be discussed and analyzed in this section. We mainly focus on unsafe impacts of these patterns, for instance, dead paths, deadlocks, and livelocks, because these impacts likely lead to potential severe anomalies [22].

4.1 Dead actors analysis

In the context of event-based systems, *dead actors* cannot be reached by any execution traces because their sets of input events are (unintentionally) empty or never emitted by any other actors. As a result, dead actors will lead to *dead execution paths* if they are not the final steps.

Algorithm 1 Analysis of dead actors for a pattern p on a system $\mathcal{S}$

```
1:  Input: Event-based system S(A, E) and a pattern p
2:  Output: Two sets of dead actors due to empty inputs R_∅ or
    missing inputs R_mx
3:  function AnalyzeDeadActors(S, p)
4:      R_∅ ← ∅
5:      R_mx ← ∅
6:      T ← ∅    ▷ a temporary set of actors that have no outputs
7:      ApplyPattern(S, p)
8:      for each x ∈ A do
9:          if (x• = ∅) then
10:             T ← T ∪ x
11:         end if
12:         if (•x = ∅) then
13:             R_∅ ← R_∅ ∪ x
14:         else
15:             x_missing_inputs ← •x
16:             for all y ∈ A \ (T ∪ x) do
17:                 x_missing_inputs ← x_missing_inputs \ y•
18:                 if (x_missing_inputs = ∅) then
19:                     break
20:                 end if
21:             end for
22:             if (x_missing_inputs ≠ ∅) then
23:                 R_mx ← R_mx ∪ x
24:             end if
25:         end if
26:     end for
27:     return R_∅ and R_mx
28: end function
```

We develop an algorithm for analyzing actors and their ports to determine a set of actors that have empty input events or some of inputs might never be emitted by any other actors (see Algorithm 1). The outcome of the algorithm can help the developers to identify the causes of potential dead execution paths and alter the actors accordingly. The statement APPLYPATTERN$(\mathcal{S}, p)$ in line 7 represents the application of a particular change pattern. If APPLYPATTERN$(\mathcal{S}, p)$ is omitted, the Algorithm 1 can be used at any time (e.g., before or after applying a certain change pattern) to find dead actors. We note that the first execution of the analysis can be costly because each pair of actors' inputs and outputs is compared. However, our evaluation shows that performing the analysis incrementally can reduce the cost notably starting from the second iteration.

Apart from dead actors, livelocks and deadlocks are also unde-

sired anomalies in any software systems but they likely happen especially in case the systems evolve without a thorough understanding and analysis of the systems and the impact of the implemented changes. Checking deadlocks and livelocks in an event-based system is similar to other distributed systems. That is, it requires complex formal specifications and model checking techniques [2] that are beyond the scope of this paper and will be part of our future endeavors. Nonetheless, in another ongoing work, we are developing a technique for mapping the snapshots of a DERA-based system onto existing formalisms for distributed systems, such as Petri Nets [14], CSS [9], or π-calculus [12]. In this way, we can enable the checking for deadlock and livelocks before or after implementing a certain system evolution using primitive actions or change patterns.

4.2 Estimation of Change Pattern Impact

Normally, in order to understand the effect of a certain change in traditional event-based systems, the developers have to investigate the source code to gather the sources and targets of the events exchanged among the components of the system. By leveraging the advantage of proposing well-defined interfaces of actors, we can better assist the developers in this task. Algorithm 2 is developed to help the developers to estimate the set of elements influenced by a change, and therefore, to determine the boundary of the region for further analysis and investigation. The notation $p.x$ denotes the corresponding parameter x of the pattern p. The outcome of Algorithm 2 comprises two sets of actors $\mathcal{R}_{in}$ and $\mathcal{R}_{out}$ representing the actors that have inputs and outputs relating to the actor x, and therefore, being influenced when x is altered, for instance, by using the change pattern.

Algorithm 2 Estimating the impact of a change pattern p applied on a system $\mathcal{S}(\mathcal{A}, \mathcal{E})$

```
1: function EstimateImpact(S, p)
2:     R_in ← ∅        ▷ R_in contains the actors whose inputs are affected
3:     R_out ← ∅       ▷ R_out comprises the actors whose outputs are affected
4:     for all a ∈ A do
5:         if (a• ∩ •p.x ≠ ∅) then
6:             R_out ← R_out ∪ a
7:         end if
8:         if (p.x• ∩ •a ≠ ∅) then
9:             R_in ← R_in ∪ a
10:        end if
11:    end for
12:    return R_in and R_out
13: end function
```

5. EVALUATION

In the scope of our work presented in this paper, a proof-of-concept implementation of the primitive actions and change patterns has been developed and incorporated into the DERA framework [25]. We conducted an evaluation to assess whether our algorithms for supporting dead actors and change impact analysis proposed in Section 3 are applicable and scalable for realistic event-based systems.

We measured the performance and scalability of the analysis algorithms with the event-based systems developed using DERA framework. The numbers of constituent elements of these systems are ranging from 50 to 1000. The estimated average numbers of events in these systems are approximately from 1.5 to 2 times of the number of the actors and the estimated average numbers of the actors' input and output ports are about 10 percent of the number of actors.

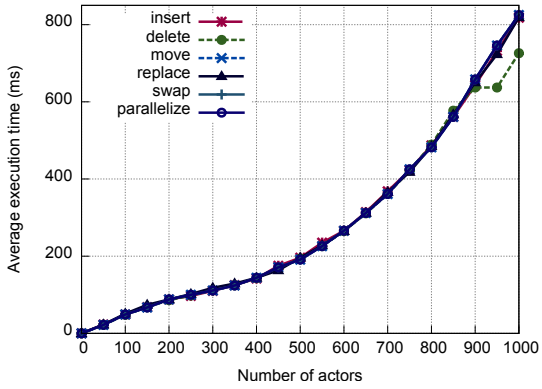


Figure 1: Performance of dead actor analysis

Using the change patterns description as proposed, also leads to an effort reduction of 11 percent [24], which means that the number of equivalent statements in comparison to programming language code reduces roughly nine folds (i.e., 1/11 %).

5.1 Performance and Scalability Evaluation

We opted to conduct the measurements on a normal desktop machine, as the analysis will usually be carried out on the workstations of the software engineers. The machine used in our experiments has an Intel 2.60GHz CPU with two gigabytes of memory running the Java VM 1.6 and a Linux operating system. Each measurement is iterated 100 times and the resulting execution time, in milliseconds, is calculated on average. We mainly report here the average numbers because the deviations are very small.

We present in Figure 1 the average execution time, in millisecond, measured for each change pattern applied on different numbers of actors. In Figure 1, the difference between the execution times of the patterns is very small. It means that the pattern plays no real role regarding the dead actor analysis performance. This is because we measure the first execution time of each pattern applied on a particular number of actors. As we discussed in Section 4, the first execution of the dead actor analysis algorithm can be costly because pairs of actors are mutually compared. Nevertheless, the exclusion of unnecessary actors at line 16 of Algorithm 1 partially reduces the execution time. We note that the execution of our change patterns only takes a fraction of a second even in the worst case, i.e., the system under consideration consists of roughly 1000 actors and 2000 event types.

In reality, the dead actors analysis can be performed in an incremental manner. That is, a thorough comparison of pairs of actors will only be carried out when the developers start analyzing or deploying the very first change in the system. In the next application of change primitive actions or change patterns, we only need to consider a subset of actors that belong to $\mathcal{R}_{\varnothing}$ and $\mathcal{R}_{mx}$ (cf. Algorithm 1) as well as the actor that is added, removed or replaced, and few related actors. Figure 2 shows our evaluation of the incremental dead actors analysis for a system consisting of 1000 actors and about 2000 event types. We can see that, start from the second iteration, the execution time has been notably reduced and then almost remains unchanged afterwards.

The average execution time of the impact analysis algorithm is

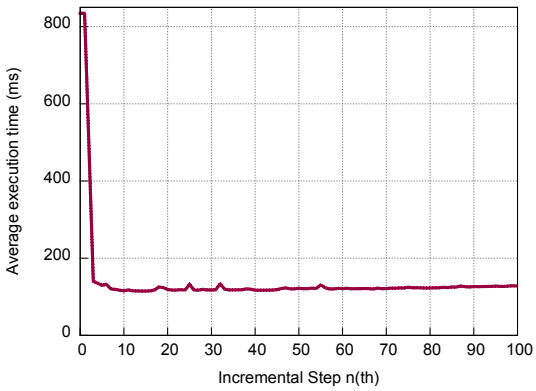


Figure 2: Performance of incremental dead actor analysis

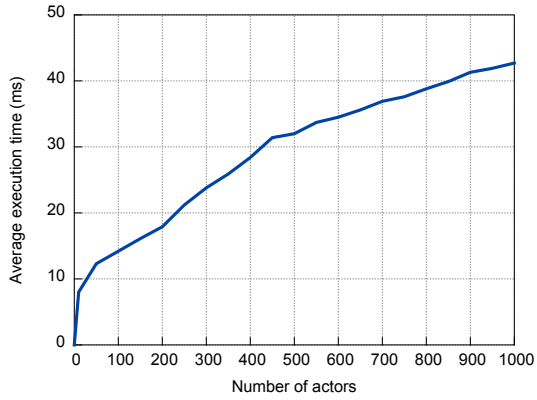


Figure 3: Performance of impact analysis

presented in Figure 3. As the algorithm only needs one iteration (see Line 4 of Algorithm 2) over the number of actors, it takes less than 50 milliseconds even in the worst case, i.e., for the systems consisting of 1000 actors and about 2000 event types.

In summary, the evaluation results of the algorithms show that our supporting techniques for dead actor and change impact analysis are considerably fast (taking less than a fraction of a second) and scalable enough (handling 1000 of elements) to be used by the software engineers in their normal working stations. The performance of our algorithms can be optimized further in several ways. For example, during the first round of execution, we retrieve the dependencies among the actors and store the dependency information in memory or in a database. As a result, the next inquiries of the dependencies between actors and matching of their inputs and outputs will be faster because we just query the dependencies in the database instead of mutually comparing pairs of actors to find out the dependencies.

6. RELATED WORK

Weber et al. [26, 21] 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 [8, 19, 20]. 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. We also investigated that the set of change patterns for PAISs can not be congruently mapped to event-based systems. For instance, the pattern **INSERT** for PAISs may insert a new component either serial, parallel or conditional [26]. In event-based systems, inserting a component will be parallel by default, since the relation to other components is not known. By regarding the interfaces of components in event-based systems, different variations of a change patterns are possible. For instance, there is only one scenario for moving a component in PAISs. In event-based systems, moving a component may have different variations of side effects, depending on how the interfaces of the involved components will be changed (do not change either input or output interface, or merge the interfaces). Therefore, change patterns for PAISs are not readily applicable for event-based systems where components are highly decoupled from each other.

Based on the formal definition of change patterns, we are able to calculate the impact of a planned change. There are a rich body of work focusing on extracting the dependency information to support analyzing the impact of a certain change [11]. Unfortunately, they often assume explicit invocations between elements, and therefore, are not readily applicable for event-based systems. The only technique to extract implicit invocation information from an event-based system, proposed by Murphy et al. [15], is Lexical Source Model Extraction (LSME). However, the results are imprecise and incomplete. Another approach to analyze event-based system is proposed by Jayaram et al. [10], which aims at extracting type information and dependencies at compile time, based on EventJava [7]. Analysis at runtime, or after changes applied, are not supported. Program slicing techniques [23, 3] can help to derive the implicit dependencies by analyzing inputs and outputs of the invocations in the source code.

While all of these techniques are powerful and promising, they can not be applied for systems that do not have their source code available, for instance, third-party libraries and components. Our approach does not depend on the availability of the system's source code. The extra cost required by our approach is for explicitly exposing the inputs and outputs of the constituent components. Nevertheless, there is no extra cost when the event-based systems are developed using the DERA framework.

The most closely related work on supporting impact analysis for event-based systems is a technique, namely, Helios, based on message dependence graphs presented by Popescu et al. [18]. Helios requires that the underlying systems must satisfy three constraints, including a message-oriented middleware supporting standard message source and sink interfaces for each component, the use of object-oriented programming languages with strong static typing, and the use of type-based filtering that supports mapping message types to programming language types as well as type-safe communication. Our prerequisites of the underlying event-based systems are less strict than Helios and easy to be satisfied by existing event-based systems. Moreover, we introduce appropriate abstractions and techniques for supporting the developers in analyzing and performing different types of changes on an event-based system.

Since all of the existing approaches for impact analysis for event-based systems need design-time information, they are not able to take a system's state at runtime into account when it comes to enact a change. For instance, deleting a component at runtime will have

no impact at all if it was already processed and there is no chance to be executed again in future. Our approach enables impact analysis on runtime information by observing and assessing event traces.

7. 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 implementing particular changes on an event-based system and analyzing the impact of these changes. We propose an approach to support change impact analysis for event-based systems through algorithms for detecting anomalies and estimating the impact of a certain change, based on change patterns that are frequently supported and used in several information systems nowadays along with their formal descriptions. The evaluation of our proof-of-concept implementation shows that our approach is feasible and applicable in realistic large-scale settings with a considerable amount of constituent elements both with respect to our approach's performance and scalability. A limitation is the performance of the dead actor analysis for very large numbers of actors, which can be mitigated to a large extent through incremental analysis, as shown in our evaluation.

Acknowledgment. This work was partially supported by the EU's Seventh Framework Programme Project INDENICA (`http://www.indenica.eu`), Grant No. 257483 and the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF), Grant No. ICT12-001.

8. REFERENCES

[1] F. Arbab and C. L. Talcott, editors. *5th Int'l Conf. Coordination Models and Languages*, volume 2315 of *LNCS*. Springer, 2002.

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[3] D. Binkley and M. Harman. A survey of empirical results on program slicing. volume 62 of *Advances in Computers*, pages 105 – 178. Elsevier, 2004.

[4] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[5] M. Broy and E.-R. Olderog. Trace-Oriented Models of Concurrency. In J. Bergstra, A. Ponse, and S. Scott, editors, *Handbook of Process Algebra*, pages 101–195. Elsevier Science B.V., 2001.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans Comput Syst*, 19(3):332–383, Aug. 2001.

[7] P. Eugster and K. R. Jayaram. Eventjava: An extension of java for event correlation. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 570–594, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] A. Hallerbach, T. Bauer, and M. Reichert. Capturing variability in business process models: the provop approach. *J. Softw. Maint. Evol.*, 22:519–546, Oct. 2010.

[9] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Apr. 1985.

[10] K. R. Jayaram and P. Eugster. Program analysis for event-based distributed systems. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS '11, pages 113–124, New York, NY, USA, 2011. ACM.

[11] S. Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 41–50, New York, NY, USA, 2011. ACM.

[12] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1st edition, June 1999.

[13] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[14] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[15] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, July 1996.

[16] S. Overbeek, M. Janssen, and P. Bommel. 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.

[17] N. W. Paton and O. Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, Mar. 1999.

[18] D. Popescu, J. Garcia, K. Bierhoff, and N. Medvidovic. Impact analysis for distributed event-based systems. In *6th ACM Int'l Conf. Distributed Event-Based Systems (DEBS)*, pages 241–251, New York, NY, USA, 2012. ACM.

[19] G. Redding, M. Dumas, A. ter Hofstede, and A. Iordachescu. Modelling flexible processes with business objects. In *IEEE Conf. on Commerce and Enterprise Computing (CEC)*, pages 41–48, 2009.

[20] M. Reichert and P. Dadam. Enabling adaptive process-aware information systems with ADEPT2. In *Handbook of Research on Business Process Modeling*, pages 173–203. Information Science Reference, 2009.

[21] S. Rinderle-Ma, M. Reichert, and B. Weber. On the formal semantics of change patterns in process-aware information systems. In *27th Int'l Conf. on Conceptual Modeling (ER)*, pages 279–293. Springer-Verlag, 2008.

[22] J. Sifakis. Deadlocks and livelocks in transition systems. In P. Dembinski, editor, *Mathematical Foundations of Computer Science 1980*, volume 88 of *LNCS*, pages 587–600. Springer Berlin Heidelberg, 1980.

[23] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, 1994.

[24] S. Tragatschnig, H. Tran, and U. Zdun. Change patterns for supporting the evolution of event-based systems. In *21st International Conference on COOPERATIVE INFORMATION SYSTEMS (CoopIS 2013)*, pages 1–8, Graz, Austria, September 2013. Springer.

[25] H. Tran and U. Zdun. Event-driven actors for supporting flexibility and scalability in service-based integration architecture. In *20th Int'l Conf. Cooperative Information Systems (CoopIS)*, pages 164–181. Springer, 2012.

[26] B. Weber, S. Rinderle, and M. Reichert. Change patterns and change support features in process-aware information systems. In *19th Int'l Conf. Advanced Information Systems Engineering (CAiSE)*, pages 574–588. Springer-Verlag, 2007.