2015 IEEE 24th International Conference on Enabling Technologies: Infrastructures for Collaborative Enterprises

Modeling Change Patterns for Impact and Conflict Analysis in Event-Driven Architectures

Simon Tragatschnig and Uwe Zdun
Research Group Software Architecture
University of Vienna, Austria
Email: {simon.tragatschnig, uwe.zdun}@univie.ac.at

***Abstract*—In distributed event-driven architectures, components are composed in a highly decoupled way, facilitating high flexibility, scalability and concurrency of distributed systems. However, the intrinsic loose coupling of its components make relations hard to identify making it challenging to analyze, maintain, and evolve an event-based architecture. For understanding the evolution of an event-based architecture, we require knowledge about its components' dependencies, which is often hard to gain due to the absence of explicit information about these dependencies. Furthermore, assisting techniques for analyzing the impacts of certain changes are missing, hindering the implementation of changes in event-driven architectures. We present in this paper a novel approach providing models to describe changes in event-based architectures on different levels of abstraction. The explicit definition of a change enables various types of analysis to increase the quality of the evolving event based systems architecture, like invalid access analysis, dangling actors analysis, change impact analysis, and dead actor analysis.**

I. Introduction

Distributed event-driven architectures are a promising solution for developing distributed systems that facilitates high flexibility, scalability, and concurrency [1], [2]. A distributed event-driven architecture consists of a number of computational or data components that communicate with each other by emitting and receiving events [2]. Each component may independently perform a particular task, for instance, accessing a database, checking a credit card, or interacting with users. However, the intrinsic loose coupling of its components make relations hard to identify and therefore it is challenging to analyze, maintain, and evolve an event-based architecture. This paper addresses supporting the evolution of distributed event-driven architecture based on models to express changes on different levels of abstraction.

As requirements on software systems evolve over time, they have to be constantly maintained and changed [3]. More than one quarter of coding time is spent on implementing changes and investigating their impact [4]. Event-based architectures are often changed on a low level of abstraction by manipulating the source code. The implementation of a particular change on an architectural level involves defining the relevant actions (e.g., adding, removing, 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 architecture, the software engineers have to deal with many technical details at different levels of abstraction, which is very tedious and error-prone.

In our work, we investigate and adapt change patterns in the context of event-based architectures dealing with the lack of prescribed execution descriptions as well as constituent elements of a system and their relationships can be arbitrarily changed at any time. Section II describes our model-based approach of describing changes on different levels of abstraction. Based on these models, it is possible to perform different novel kinds of impact and conflict analysis, like invalid access analysis, dangling actors analysis, change impact analysis, and dead actor analysis, as described in Section III. Our approach is then discussed in Section IV.

II. Modeling Change Patterns

The main focus of our work is to support changes for event-driven architectures. Without loss of generality, we adopt the notion that a generic event-based architecture comprises a number of components performing computational or data tasks and communicating by exchanging events through event channels [2].

Maintaining event-based architectures is challenging because of the absence of explicit information on the dependencies of its components, which may have been extracted from source code. Providing models to specify event-based architectures allows developers to focus on their concepts, like events and event emitting or consuming components. Basis for modeling changes is a model of the event-based architecture, on which additional tooling can be built upon to support the maintainers. For demonstration purpose, we use the DERA framework [5] that provides this basic concepts for modeling and developing event-based architectures.

The DERA Meta Model, proposed in [5], [6], describes the basic concepts and can easily be generalized to the concepts found in many other event-based architectures. In DERA, a component is represented by an *event actor* (or *actor* for short), which represents a computational or data handling unit. For instance, this may be the executing a service invocation, or accessing and transforming data. An *event* can be considered essentially as "any happening of interest that can be observed from within a computer" [2] (or a software system). DERA uses the notion of *event types* to represent a class of events that share a common set of attributes. *Actors* provide two ports, the *input* and the *output port*. A *port* describes the interface of an actor. Instances of the defined *event types* in the *input port* will trigger the actor. The actor will emit instances of event types defined in its *output port* when it finishes execution. *Execution domains* encapsulate a logical group of related actors.

978-1-4673-7692-1/15 $31.00 © 2015 IEEE
DOI 10.1109/WETICE.2015.13

To deal with the complexity and the large degree of flexibility of event-based architectures, we propose expressing changes on different levels of abstraction. On the lower abstraction level, change primitives are used to express fine granular changes on the modeled DERA application. This level is used by our system to execute a change. On the next higher level, change patterns are used to express changes of the system, for instance moving an actor. Change patterns are transformed into a set of change primitives, which can be applied to a DERA application. On the highest level of abstraction, change pattern descriptions define syntax and semantics of change patterns. A model for change patterns can be generated from this change pattern descriptions.

Change Primitives - We use low-level primitives, called change primitives introduced in [7], for encapsulating the basic change actions for populating and modifying event-based architectures, such as adding or removing an event or an actor, replacing an event or actor, and so forth. We already implemented the proposed set of primitives for our DERA prototype. However, we still have to provide a model and tooling for developers to express change primitives.

Change Patterns - Change patterns for event-based architectures support software engineers to describe and apply desired changes at a higher level of abstraction. They are defined based on the patterns that are frequently occurring and supported in most of today's information systems according to the survey presented by Weber et al [8].

A change pattern basically expresses that a set of actors should change it's position within a DERA application, related to other implicitly dependent actors with matching interfaces. We need to define a model to express a specific change pattern. As this represents a high level change, it can be transformed in a sequence of change primitives applied by the runtime environment.

Change Pattern Description - As the possible spectrum of change patterns is broad, a predefined language to express changes is not a sufficient tool because it would have to be adapted for each additional change pattern or pattern variants. Therefore, there is a need to model change patterns and their impact - the Change Pattern Descriptions. They allow a change pattern developer to define and modify her own set of change patterns and its semantics, using set operated statements like union, intersection, etc.

III. Impact and Conflict Analysis

Applying a change in event-based architectures may cause undesired side effects. To assist software engineers detecting these side effects, modeled changes on different levels of abstraction enables calculating the impact and possible conflicts of a specific change on a running DERA application. In this section we present some analysis.

Invalid Access Analysis - Applying a set of changes to a system may cause conflicts in the sequence of change primitives, like referencing actors which were deleted previously. We identified three scenarios which cause invalid access of DERA elements: (1) add/remove an actor, which was already deleted, (2) reference a port of an actor, which was already deleted, (3) remove an event from an actor's port which was already removed. Algorithm 1 shows pseudocode for identifying invalid access. As trying to delete an already deleted element has no effect, referencing or adding an already deleted element may be indicating conflicts. Adding an already deleted actor to the DERA application is not a conflict, but may be a hint that some change patterns are conflicting.

Algorithm 1 Invalid Access Analysis

```
1:  removedElements = all primitives removing actors
2:  removedElements += all primitives removing events
3:  for all primitive statements as statement do
4:      for all removedElements as removedElement do
5:          if statement references removedElement then
6:              if statement is located after removedElement then
7:                  Show error
8:              end if
9:          end if
10:     end for
11: end for
```

Dangling Actors Analysis - In the context of event-based architectures, *dangling actors* can not trigger other actors or cause the application's end because they do not emit output events which are defined as input events of other actors. This is not a problem in general, but if an actor becomes a *dangling actor* directly after applying a change, this situation may not be intended. To determine *dangling actors* caused by changes, we have to simulate the changes and analyze the results. The algorithm listed in Algorithm 2 compares each actor's input port with each other actors' output ports. If there is no match for the actor, this actor is dangling.

Algorithm 2 Dangling Actor Analysis

```
1:  allActors = all existing actors
2:  changedActors = all actors affected by primitives
3:  for all changedActors as actor1 do
4:      for all allActors as actor2 do
5:          isDangling = true
6:          if actor2 : output equals actor1 : input then
7:              isDangling = false
8:              break
9:          end if
10:     end for
11:     if isDangling then
12:         danglingActors += actor1
13:     end if
14: end for
```

Change Impact Analysis, proposed in [9], allows developers to identify elements being modified by a certain change without investigating dependency information from code. This supports developers to determine the boundary of the region for further analysis and investigation.

Dead Actor Analysis, proposed in [9], supports the developers by identifying actors which are unreachable, which means that no other actor will emit the event types expected by the unreachable actors. This is an indication that a change may cause unexpected side effects.

IV. Discussion and Future Work

Maintaining event-based architectures is challenging because of the absence of explicit information on the dependencies of its components. Applying changes may cause unwanted side effects which are difficult to perceive due to missing explicit dependency information. We address this challenge in

our work by introducing models to express changes on different levels of abstraction. Based on this models it is possible to run algorithms to predict a change's approachimpact and detect anomalies or conflicts.

Weber et al. [8], [10] 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 [11]–[13]. Event-based architectures, 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, the semantics of the change pattern is different in event-based architectures. Using our proposed model to describe change patterns, we work on a catalog defining the semantics of change patterns for event-based architectures.

Based on the definition of change patterns' semantics, 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 [14]. Unfortunately, they often assume explicit invocations between elements, and therefore, are not readily applicable for event-based architectures.

Techniques to extract implicit invocation information from event-based architectures are based on source code. For instance, Lexical Source Model Extraction (LSME) [15], program slicing techniques [16], [17], extracting type information and dependencies at compile time [18]. Analysis at runtime, or after changes applied, are not supported. The most closely related work on supporting impact analysis for event-based architectures is a technique, namely, Helios, based on message dependence graphs presented by Popescu et al. [19]. Helios requires that the underlying systems satisfy some constraints, including a message-oriented middleware supporting standard message source and sink interfaces for each component. Our prerequisites of the underlying event-based architectures are less strict than Helios and easy to be satisfied by existing event-based architectures. Also, we introduce appropriate abstractions and techniques for supporting the developers in analyzing and performing different types of changes on an event-based architecture. 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. In contrast, our work can extract all needed information either from a model or from meta data of running instances. The extra cost required by our approach is for explicitly exposing the in- and outputs of the components. There is no extra cost when the event-based architectures are developed using the DERA framework.

As future work we plan to realize model-based domain specific languages and tooling to express changes on the proposed levels of abstraction. As a result of our work, the architect of a distributed event-driven architecture should be able to design, change, or maintain systems benefiting from the key benefits regarding the architectural qualities of event-driven architectures such as high flexibility, scalability, and concurrency, on the one hand. On the other hand, the architect can rely on tools that tame the loosely coupled nature of these architectures and make them manageable and analyzable.

REFERENCES

[1] L. Fiege, G. Mühl, and F. C. Gärtner, "Modular event-based systems," *Knowl. Eng. Rev.*, vol. 17, no. 4, pp. 359–388, Dec. 2002.

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

[3] M. M. Lehman, "On understanding laws, evolution, and conservation in the large-program life cycle," *J. Syst. Softw.*, vol. 1, pp. 213–221, Sep. 1984.

[4] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10, 2010, pp. 185–194.

[5] 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)*. Springer, 2012, pp. 164–181.

[6] ——, "Event Actors Based Approach for Supporting Analysis and Verification of Event-Driven Architectures," in *17th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, Sep 2013, pp. 217–226.

[7] 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)*. Graz, Austria: Springer, September 2013, pp. 283–290.

[8] 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)*. Springer-Verlag, 2007, pp. 574–588.

[9] S. Tragatschnig, H. Tran, and U. Zdun, "Impact analysis for event-based systems using change patterns," in *29th Symposium On Applied Computing (SAC 2014) - Cooperative Systems*. ACM, March 2014.

[10] 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)*. Springer, 2008, pp. 279–293.

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

[12] 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)*, 2009, pp. 41–48.

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

[14] 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*, ser. IWPSE-EVOL '11. New York, NY, USA: ACM, 2011, pp. 41–50.

[15] G. C. Murphy and D. Notkin, "Lightweight lexical source model extraction," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 3, pp. 262–292, Jul. 1996.

[16] F. Tip, "A survey of program slicing techniques," Amsterdam, The Netherlands, Tech. Rep., 1994.

[17] D. Binkley and M. Harman, "A survey of empirical results on program slicing," ser. Advances in Computers. Elsevier, 2004, vol. 62, pp. 105 – 178.

[18] 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*, ser. DEBS '11. New York, NY, USA: ACM, 2011, pp. 113–124.

[19] 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)*. ACM, 2012, pp. 241–251.