Domain Specific Languages for Maintaining and Analyzing Changes in Event-Based Architectures

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

***Abstract*—A main characteristic of event-driven architectures is that components are highly decoupled, which facilitates high flexibility, scalability and concurrency of distributed systems. This intrinsic loose coupling of components introduces the challenge to identify dependencies between the components, which have to be known to developers to analyze, maintain, and evolve an event-based architecture. The knowledge about component's dependencies 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. In this paper we present a novel approach to support developers in evolving event-based architectures by using model-based domain specific languages for describing changes at different levels of abstraction. The DSLs' models are used to support analysis of specific changes to increase the quality of the evolving event based systems architecture.**

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 makes 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.

Software systems often have to evolve over time due to changing requirements. Therefore, 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]. By analyzing evolution of software systems, Weber et al. identify a set of change patterns that recur in many of existing software systems [5]. 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 cannot be changed during runtime or just slightly deviated from the initial schema [6]–[8]. As a result, these approaches are not readily applicable for event-based architectures where components are highly decoupled and the dependencies between components can be 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.

Event-based architectures are often changed at a low level of abstraction by manipulating the source code. However, the implementation of a particular change always have to take the consequences into account, as other components might be affected by this change. 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 previous work [12] we aimed to support software engineers to describe and apply desired changes at a higher level of abstraction. Therefore, we investigated and adapted the change patterns in the context of event-based architectures dealing with the lack of prescribed execution descriptions and considering arbitrarily changing relationships of constituent elements of a system. Based on the notion of change patterns defined by Weber et al [5], we presented fundamental abstractions for implementing certain changes in an event-based architecture [11]. As an extension of this work, in order to deal with the complexity and the large degree of flexibility of event-based architectures, we aim at supporting system evolution at different levels of abstraction. In this paper, we present domain specific languages (DSLs) at different levels of abstraction to express changes. The implementation of this approach is based on our DERA prototype [11] for purpose of demonstration. But our results can be generalized to other kinds of event-driven architectures easily, as only the very basic concepts of the DERA meta-model, present in most event-driven architectures, are used in the approach presented in this paper.

In our approach, a specific change at the lowest level of abstraction can be expressed by a sequence of primitive change operations [11], such as adding or removing an event or an actor, replacing an event or actor, and so forth. As these primitive changes will directly modify features of a running instance in the event-based architecture, they are not comfortable to use for developers. Therefore, we present a DSL to express change patterns at a higher level of abstraction, which are automatically transformed into a set of primitive change operations using model transformations. As the demands for the set of change patterns will evolve over time, we also propose another DSL to model the change patterns themselves.

Using our model-based DSLs to express changes allows developers to calculate dependencies between the modified components of the event-based architecture and to run analyses to calculate a change's impact or predict conflicts. This way, the architect can benefit in the architectural design and

978-1-4673-9331-7/15 $31.00 © 2015 IEEE
DOI 10.1109/EDOCW.2015.25

evolution of a distributed event-driven architecture from its key benefits on architectural qualities such as high flexibility, scalability, and concurrency, but still rely on tools that tame the loosely coupled nature of these architectures and make them manageable and analyzable.

Section II explains the background on event-driven architectures and the generalizable concepts from the DERA metamodel we used as basis of our prototype implementation. A running example of an evolving DERA application is presented in Section III. Section IV describes our model-based approach for describing changes at different levels of abstraction. An overview of related work is presented in Section V. Our lessons learned are discussed in Section VI. We conclude in Section VII.

II. Background

A. Event-driven Architectures

The main focus of our work is to support changes in 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]. Due to the inherent loosely coupled nature of the participating components of an event-based architecture, understanding and implementing changes is challenging for software engineers. To support better applying specific changes and understanding their impacts, we slightly reduced the non-determinism due to the loosely coupled 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.

A component's interface can be altered (adding/removing event types). Therefore, the input/output event information can only be observed at a certain point in time. The major advantage of the exposed interface 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 in case of using third-party libraries and components) [2].

For demonstration purpose, we build upon the DERA framework [9] that provides basic concepts for modeling and developing event-based architectures and supports the three requirements. The DERA Meta Model is described more detailed in the following Section II-B.

B. DERA Meta Model

The DERA Meta Model describes the basic concepts of event-based architectures along the lines outlined in the previous section and can thus easily be generalized to the concepts found in many other event-based architectures. Figure 1 shows a simplified excerpt of the DERA Meta Model depicting only its very basic concepts. Due to space reasons and to decrease complexity, in this paper, we focus on the core concepts of DERA. A detailed description of DERA can be found in [9]. 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 may emit instances of event types defined in its *output port* when it finishes execution. This causes an implicit control flow, defined by a matching set of event types of an output port of one actor and an input port of another actor. Note, that there exist several types of DERA actors [9], differing slightly in its behavior. For the sake of simplicity we do not explain them in detail. DERA applications are organized in *execution domains*, which encapsulate a logical group of related actors. 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 [9].

A DERA Model, which is an instance of the DERA Meta Model, describes a specific application for an event-based architecture and can be executed within the DERA runtime environment. Figure 2 shows a simple DERA Model. Each actor is depicted as a rectangle, which has an input port on the left and an output port on the right, holding the event types to be received or emitted. The dashed arrows show the implicit control flow which is defined by matching event types of input- and output-ports.

We call an actor B a *successor* of actor A, if the event types defined in the input port of B matches the event types in the output port of A. In return, A is a *predecessor* of B. This means, that a *successor* is executed after the execution of its *predecessor* has finished. For instance, *Book Car* is a successor of *Book Hotel*, and *Book Flight* is the predecessor of *Book Hotel*.

III. Illustrative Example

We illustrate the complexity of a change within a simplified event-based application for processing bookings for flight, hotels and car rentals. The simplistic initial architecture of our event-based application is depicted in Figure 2, whereas Figure 3 shows the textual representation of this DERA Model. A box represents an actor, showing the input port on the left side and the output port at the right side. The implicit control flow, defined by a matching set of event types of an output port of one actor and an input port of another actor, is shown as a dashed arrow.

After receiving an itinerary request from a customer, expressed by the actor *Receive Itinerary Request*, the event *initialized* is emitted. This event encapsulates the data of the itinerary. The actor *Book Flight* receives this event, interprets

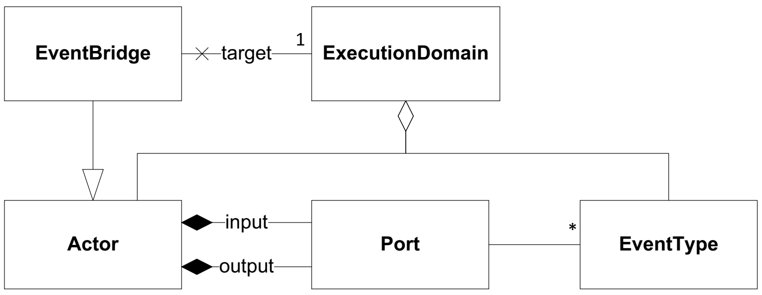

Fig. 1. Simplified Excerpt of the DERA Meta Model (only the necessary elements needed for this paper)

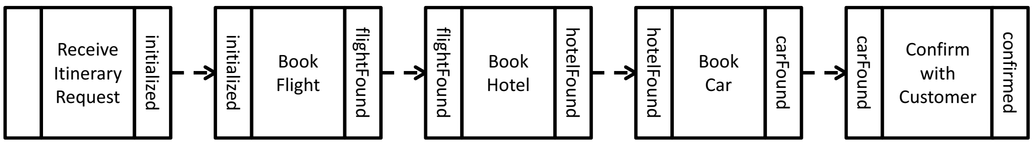

Fig. 2. Initial architecture of the travel booking application

```
module travelBookingApp
Event initialized
Event flightFound
Event hotelFound
Event carFound
Event confirmed
EventActor ReceiveItineraryRequest
           input [] output [initialized]
EventActor BookFlight
           input [initialized] output [flightFound]
EventActor BookHotel
           input [flightFound] output [hotelFound]
EventActor BookCar
           input [hotelFound] output [carFound]
EventActor ConfirmWithCustomer
           input [carFound] output [confirmed]
```

Fig. 3. Textual DERA Model of the travel booking application depicted in Figure 2

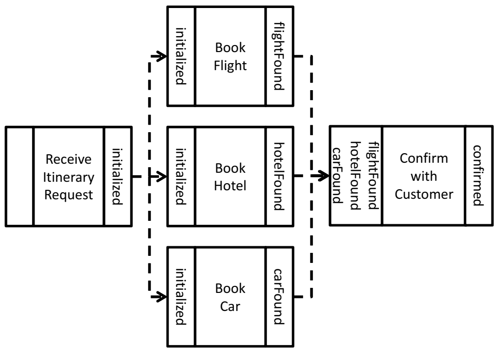

Fig. 4. Travel booking application with parallelized actors

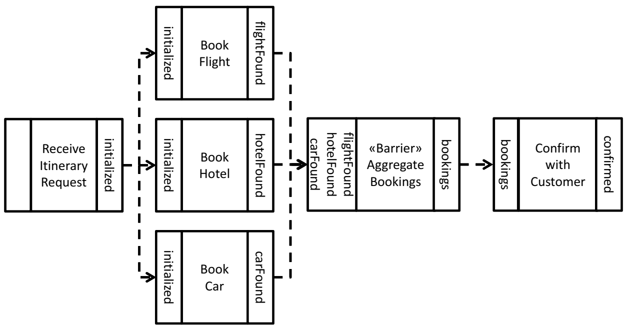

Fig. 5. Travel booking application with parallelized and joined actors

the itinerary and tries to find a suitable flight. When a flight is found, the event *flightFound* is emitted, which causes the actor *Book Hotel* to be executed. *Book Hotel* tries to find a suited hotel and emits the event *hotelFound*. This causes the actor *Book Car* to book a rental car, emitting the event *carFound*. This event triggers the actor *Confirm with Customer* which lets the customer approve the suggested bookings. When the customer is satisfied, the event *confirmed* is emitted.

Consider over time we want to improve the architecture of the travel booking application. In particular, in the initial system design, the sequential execution of the actors may not be appropriate, since all relevant data for the actors *Book Flight*, *Book Hotel*, and *Book Car* are already contained in the *initialized* event. An improved design would parallelize the concerned actors, as shown in Figure 4. This change causes the actors *Book Flight*, *Book Hotel*, and *Book Car* to be executed in parallel right after the *initialized* event was emitted. After an actor finished execution, the actor *Confirm with Customer* is triggered.

By the definition of the behavior of a DERA actor, this situation causes this last actor to be triggered three times, each time by a different event type. As this is not the intention, we also have to insert an additional blocking actor, waiting for the occurrence of all three events *flightFound*, *hotelFound*, and *carFound* and then triggering the last actor as shown in Figure 5. For this purpose we insert a new actor *Aggregate Bookings*, which is of a special type *Barrier* described in [9]. The barrier actor is waiting for the occurrence of all event types before executing and emitting its output events.

In event-based architectures, inserting a new actor may lead to parallel execution paths. In our example, however, the semantics of the insert is rather a serial insert, in which the transitions from the preceding actor to the succeeding actors will be strictly redirected through the new actor as described in [11]. Therefore, inserting a new actor must not lead to any loops.

To apply the desired changes, a developer has to ensure the following constraints: For parallelizing the actors *Book Flight*, *Book Hotel*, and *Book Car*, their input event types have to be changed to the output event type of *Receive Itinerary Request*. Also, the input port of *Confirm with Customer* must contain all event types of the actors to be parallelized. To insert the new actor *Aggregate Bookings* it must be ensured that the parallelized actors' input port does not contain any event types causing loops by the new actor's output port. The input port of *Aggregate Bookings* must match to the parallelized actors' output ports. The input port of *Confirm with Customer* now must not contain event types of the paralleized actors' output ports, but must contain event types of *Aggregate Bookings*.

Although the desired changes seem to be quite simple at a high level of abstraction, the developer has to deal with complex constraints on a lower level of abstraction, always having to take care for all affected actors and their dependencies and reevaluating the situation every time a port gets changed. As dependency information is implicitly defined by matching event types between input and output ports, the developer has to extract this information directly from the source code or the application's models. When changes on a running application are applied, the dependency information shown in the source code may even be outdated.

The illustrative example reveals that manually applying changes to an event-based architecture is error prone and extremely challenging for developers. In the following section we will resolve this example at different levels of abstraction and show how developers can be supported to manage such changes more easily.

IV. Modeling Change Patterns

Maintaining an event-based architecture is challenging because of the absence of explicit information on the dependencies of its components. Therefore, knowledge about the dependencies between components has to be extracted from the source code. Assisting techniques for analyzing the impacts of certain changes are missing, hindering the implementation of changes in event-based architectures. Using a model-based domain specific language (DSL) for specifying event-based architectures allows developers to focus on its concepts, like events and event emitting or consuming components. Basis to express changes with a DSL is a model of the event-based architecture, upon which additional tooling can be built to support the maintainers. Our solution rests on the DERA meta model which is briefly described in Section II-B and was proposed in [9], [10].

In our previous work [12] we proposed models to express changes at different levels of abstraction to deal with the complexity and the large degree of flexibility of event-based architecture. Based on this approach, we developed DSLs for change patterns to express changes at different level of abstraction, as shown in Figure 6.

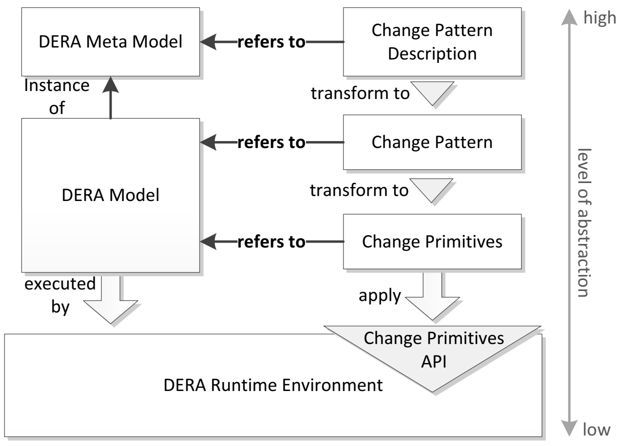

Fig. 6. Overview of the interplay of the DSLs and DERA runtime environment

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 analyze or 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 change patterns.

Using the Change Pattern Description DSL, explained in SectionIV-C, a catalog of change patterns can be modeled, which may evolve over time by modifying existing patterns or adding pattern variants. From this model of change patterns, a Change Pattern DSL is generated, as explained in Section IV-B. A developer can use this Change Pattern DSL to express a desired change for a specific event-based architecture. For our example, the actors *Book Flight*, *Book Hotel*, and *Book Car* should be executed in parallel using the pattern **Parallelize**. To join the parallel execution flow and aggregate the bookings, we need to add a Barrier actor, which can be done using the **SerialInsert** pattern. To apply these changes, they are transformed into a set of change primitives, which are expressed using our Change Primitives DSL explained in Section IV-A. The transformation from a change pattern instance to change primitives is defined in the change pattern catalog model using the Change Pattern Description DSL. The set of change primitives can then be enacted by the change primitives API implementation of the event-based architecture runtime environment.

To realize the model-based DSLs, we used the Eclipse Frameworks XText[1] to develop the DSLs, and XTend[2] to express the transformations between the different levels of abstraction.

[1]https://www.eclipse.org/Xtext/
[2]https://eclipse.org/xtend/

A. Change Primitives

We use low-level change primitives which describe simple, primitive low-level actions for populating and modifying event-based architectures that conform to the definitions we provided in Section II. The primitives used in this paper are outlined in Table I, which support adding or removing an event or an actor, replacing an event or actor, and so forth. An implementation of the proposed set of primitives is implemented for our DERA prototype.

We developed a model-based DSL to express change primitives, referencing the DERA Model described in Section II-B. Also, we provide generators to transform instances of modeled change primitives to executable code, which can be executed using the DERA prototype. The DSL offers three basic operators which can be applied to instances of specific DERA Model elements or its attributes:

=	set: assigns the right-hand set of actors or event types to the left-hand side.
+=	add: adds the right-hand set of actors or event types to the left-hand set.
-=	remove: removes the right-hand set of actors or event types from the left-hand set.

Table I shows the mapping of change primitives and the corresponding DSL expressions. Based on these primitives, in the following Section IV-B we present change patterns for event-based architectures with which the software engineers can easier describe and apply desired changes at a higher level of abstraction.

To pick up on the illustrative example described in Section III, we have to perform the following steps to calculate the proper set of change primitives. The Items 1 and 2 are parallelizing the actors *Book Flight*, *Book Hotel*, and *Book Car*, whereas Items 3 to 5 are inserting the new actor *Aggregate Bookings*. In Figure 7 the respective DSL code using change primitives is shown.

1) The input port of *Confirm with Customer* has to contain all event types of the output ports of *Book Flight*, *Book Hotel*, and *Book Car*.
2) Determine output events of *Receive Itinerary Request*, which do not intersect with output events of *Book Flight*, *Book Hotel*, and *Book Car*. The result is the new set of event types for the input port of *Book Flight*, *Book Hotel*, and *Book Car*.
3) To prevent loops, the input ports of the predecessors *Book Flight*, *Book Hotel*, and *Book Car* must not contain any event type of the intersection between the output port of *Aggregate Bookings* and input ports of *Book Flight*, *Book Hotel*, and *Book Car*.
4) The input port of *Aggregate Bookings* has to contain all event types of the output ports of all predecessors *Book Flight*, *Book Hotel*, and *Book Car*.
5) The input port of *Confirm with Customer* must not contain event types of its previous predecessors' output ports, but must contain event types of its new predecessor *Aggregate Bookings*

Pattern: SerialInsert

Description SerialInsert(x, Y, Z) will add an actor x such that all actors of Y will become predecessors and those of Z will become successors of x, transferring all dependencies between y and z to x.

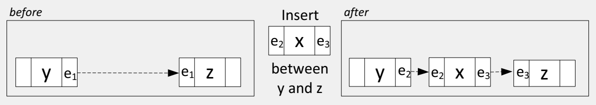

Fig. 8. Change Pattern: SerialInsert

```
Change Set TravelBooking {
    Parallelize travelBookingApp.BookFlight,
                travelBookingApp.BookHotel,
                travelBookingApp.BookCar
      between travelBookingApp.ReceiveItineraryRequest
      and travelBookingApp.ConfirmWithCustomer.


    SerialInsert travelBookingApp.AggregateBookings
      between travelBookingApp.BookFlight,
              travelBookingApp.BookHotel,
              travelBookingApp.BookCar
      and travelBookingApp.ConfirmWithCustomer.
}
```

Fig. 9. Change pattern instances for the travel booking application

B. Change Patterns

Change patterns for event-based architectures support software engineers to describe and apply desired changes at a higher level of abstraction. We use fundamental abstractions for implementing certain changes in an event-based architecture based on the notion of change patterns. In this approach, low-level primitives, explained in Section IV-A, 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 architectures are defined. Table II gives an overview about the realized change patterns.

A change pattern basically expresses that a set of actors should change it's position within a DERA application, related to other actors which might be successors or predecessors after the change is applied. A change pattern consists of various statements, which describe a change. A change defines a set of source actors, which define the actors which are the context of a change, e.g., actors to be moved, inserted or deleted. A change might also define existing or future relations to other actors.

For instance, the change pattern **SerialInsert** defined in Figure 8 can express that a source actor 'x' has to be inserted between actor 'y' and 'z', transferring all dependencies between y and z to x.

We developed a model-based Change Pattern DSL which is designed to support developers expressing changes without detailed knowledge of the DERA framework. It is linked with the DERA Model, so we can benefit from code completion and validation within an Eclipse IDE.

We provide a model-based DSL with a static set of

```
primitives TravelBooking
{
  // **** BEGIN parallelize ****
  // cleanup parallelized input ports
  travelBookingApp.BookFlight:input -= {};
  travelBookingApp.BookHotel:input -= travelBookingApp.flightFound;
  travelBookingApp.BookCar:input -= travelBookingApp.hotelFound;
  // cleanup successor input ports
  travelBookingApp.ConfirmWithCustomer:input -= travelBookingApp.carFound;
  // parallelize actors
  travelBookingApp.BookFlight:input += travelBookingApp.initialized;
  travelBookingApp.BookHotel:input += travelBookingApp.initialized;
  travelBookingApp.BookCar:input += travelBookingApp.initialized;
  // route to successors
  travelBookingApp.ConfirmWithCustomer:input += travelBookingApp.flightFound,
                                              travelBookingApp.hotelFound,
                                              travelBookingApp.carFound;
  // **** END parallelize ****

  // **** BEGIN serialInsert ****
  // prevent loops
  travelBookingApp.BookFlight:input -= {};
  travelBookingApp.BookHotel:input -= {};
  travelBookingApp.BookCar:input -= {};
  // insert elements
  // DERA model for the new actor in module travelBookingApp
  // Event bookings
  // Barrier AggregateBookings input[flightFound, hotelFound, carFound] output [bookings]
  travelBookingApp += travelBookingApp.AggregateBookings;
  travelBookingApp.ConfirmWithCustomer:input -= travelBookingApp.flightFound,
                                              travelBookingApp.hotelFound,
                                              travelBookingApp.carFound;
  travelBookingApp.ConfirmWithCustomer:input += travelBookingApp.bookings;
  travelBookingApp.AggregateBookings:input += travelBookingApp.flightFound,
                                            travelBookingApp.hotelFound,
                                            travelBookingApp.carFound;
  // **** END serialInsert ****
}
```

Fig. 7. Change primitives for changing the travel booking application

Change Primitive Specification	DSL Syntax	Description
`add(Actor a)`	*ExecutionDomain* `+= actors;`	Add the actor a to the execution domain
`remove(Actor a)`	*ExecutionDomain* `-= actors;`	Remove the actor a from the execution domain
`add(Port p, EventType[] events)`	*Actor:Port* `+= events;`	Add a set of *events* to port p
`remove(Port p, EventType[] events)`	*Actor:Port* `-= events;`	Remove a set of *events* from port p
`replace(Port p, EventType[] events)`	*Actor:Port* `= events;`	Replace all events of port p with another set of *events*

TABLE I. OUTLINE OF CHANGE PRIMITIVES AND THEIR DSL EQUIVALENT

basic change patterns (**PARALLELINSERT**, **SERIALINSERT**, **DELETE**, **MOVE**, **REPLACE**, **SWAP**, **PARALLELIZE**, **MIGRATE**), coming with generators to transform change pattern instances into a set of change primitives.

Adding other patterns or pattern variants to this set of change patterns is a lot of coding work, as the grammar as well as the transformation into primitives have to be redefined. Therefore, we developed a Change Pattern Description DSL described in Section IV-C which allows developers to specify their own set of change patterns.

Figure 9 shows the solution for the illustrative example described in Section III using the Change Pattern DSL. As the set of change patterns may evolve over time, we provide a DSL to model a catalog of change patterns which is described in the next section.

C. 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 variant. Therefore, we decided to develop a model-based DSL to describe change patterns and their impact. That is, from textual models of change pattern descriptions, we can generate the code for the Change Pattern DSL described in Section IV-B. This way, the Change Pattern DSL's definition is highly extensible and easy to change.

The pattern **INSERT** conveys an idea about pattern variants: **PARALLELINSERT** describes how to insert a new actor into the implicit execution path. One possibility is to just add a new dependency information to the ports of the preceding and succeeding actor. This may lead to a parallel execution path, causing unexpected execution behavior. Another variant of the pattern, **SERIALINSERT**, will remove all existing transitions between the preceding and succeeding actor before the new

Change Pattern	Description
PARALLEL-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
SERIAL-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, transferring all dependencies between y and z to x
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 II. AN OVERVIEW OF CHANGE PATTERNS

actor is inserted.

The Change Pattern Description DSL allows a change pattern developer to define and modify her own set of change patterns and its semantics, using set operation statements like union, intersection, etc. Results of the set operations can be stored in variables or directly assigned to an DERA model element like an actor. Instances of change pattern descriptions can be transformed to a stand-alone Change Pattern DSL including the change pattern grammar definition, generators to transform change patterns into sets of change primitives, as well as editors for the Eclipse IDE.

Figure 10 shows the definition of **SERIALINSERT** and **PARALLELIZE** used to provide a solution for the illustrative example described in Section III using the Change Pattern Description DSL. The descriptions contains a name, a short textual description, a simple syntax definition and the pattern's semantics expressed by set-based transformation rules.

The model for defining a pattern is shown in Figure 11: A *PatternCatalog* describes a set of *Patterns*. A *Pattern* describes the transformation from a source context to a target context. Source and target definitions are rather placeholder than real existing DERA elements, because at this level of abstraction there is no specific DERA element instance. The source context (*PatternSource*) describes the DERA elements to which the change should be applied, whereas the target context (*Pattern-Target*) describes the DERA elements which are related to the source context. The *TransformationDescription* defines the set-based operations which are needed to be enacted to apply a change.

The transformation description of our Change Pattern De-

```
Catalog at.ac.univie.swa.changepatterns
Pattern SerialInsert :
"Inserts an actor x between y and z,
 transferring all dependencies between y and z to x"
{
 keyword: "SerialInsert"
 from: actor toBeInserted
 to: "between" actor predecessor "and" actor successor
 transform:
    // prevent loops
    foreach predecessors as predecessor {
        predecessor:in = predecessor:in without
           (toBeInsertedSet:out intersect predecessor:in)
    }
    // insert
    foreach toBeInsertedSet as toBeInserted {
        toBeInserted:in = toBeInserted:in union
           predecessors:out
    }
    foreach successors as successor {
        successor:in = toBeInsertedSet:out union
           (successor:in without
           (predecessors:out intersect successor:in))
    }
}

Pattern Parallelize :
 "Enables concurrent execution of actors
  that where performed sequentially before"
 {
    keyword: "Parallelize"
    from: actors actorSet
    to: "between" actors predecessors
        "and" actors successors
    transform:
      // successor awaiting parallelized actors' events
      foreach successors as successor {
        foreach actorSet as anActor {
          successor:in = successor:in union
                         (anActor:out intersect actorSet:in)
        }
      }
      // predecessor triggers parallelized actors
      foreach actorSet as anActor {
        anActor:in = predecessors:out without actorSet:out
      }
 }
```

Fig. 10. Change pattern description of **SERIALINSERT** and **PARALLELIZE**

scription DSL shown in Figure 12 supports storing results of set operations (*Expressions*) in variables (*VariableDefinition*) or directly into DERA elements *ValueAssignment*. Also, iteration over a set, e.g. a set of DERA elements, can be realized with *ForEach* The most simple *Expression* is a reference to a DERA element or a variable (*SetOperationStatementElement*). The *SetOperationStatement* can be used to apply set operations to sets of DERA elements. A *SetSelector* can be used to specify a more complex set of DERA elements.

V. RELATED WORK

Weber et al. [5], [13] 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 [14]–[16]. 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

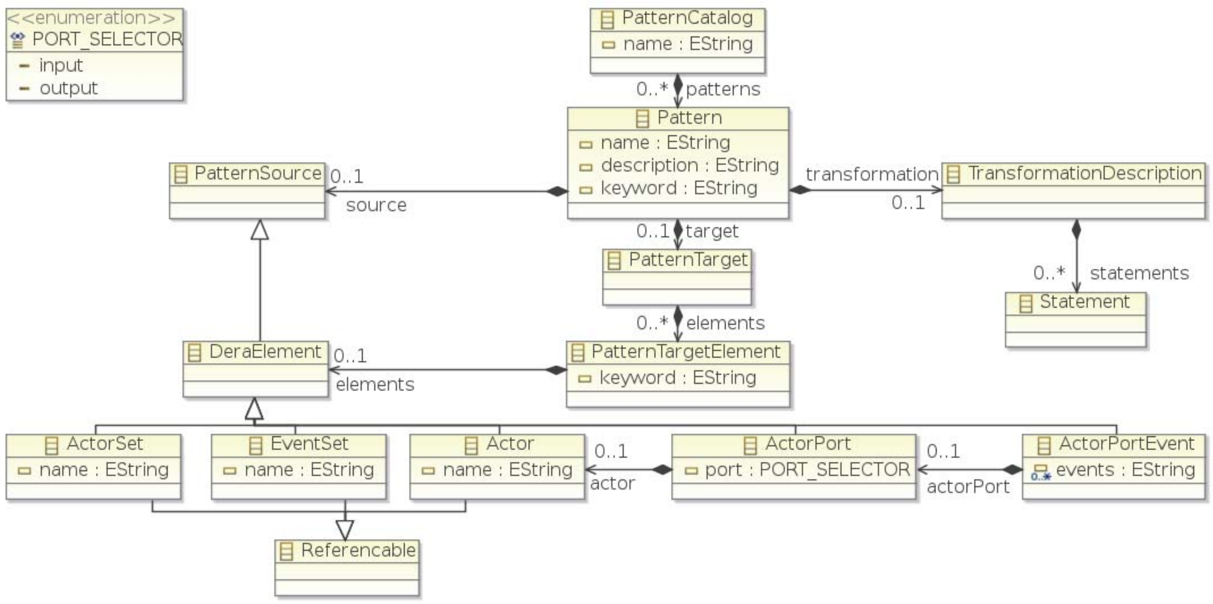

Fig. 11. Outline of the Change Pattern Description DSL Model - Pattern Description

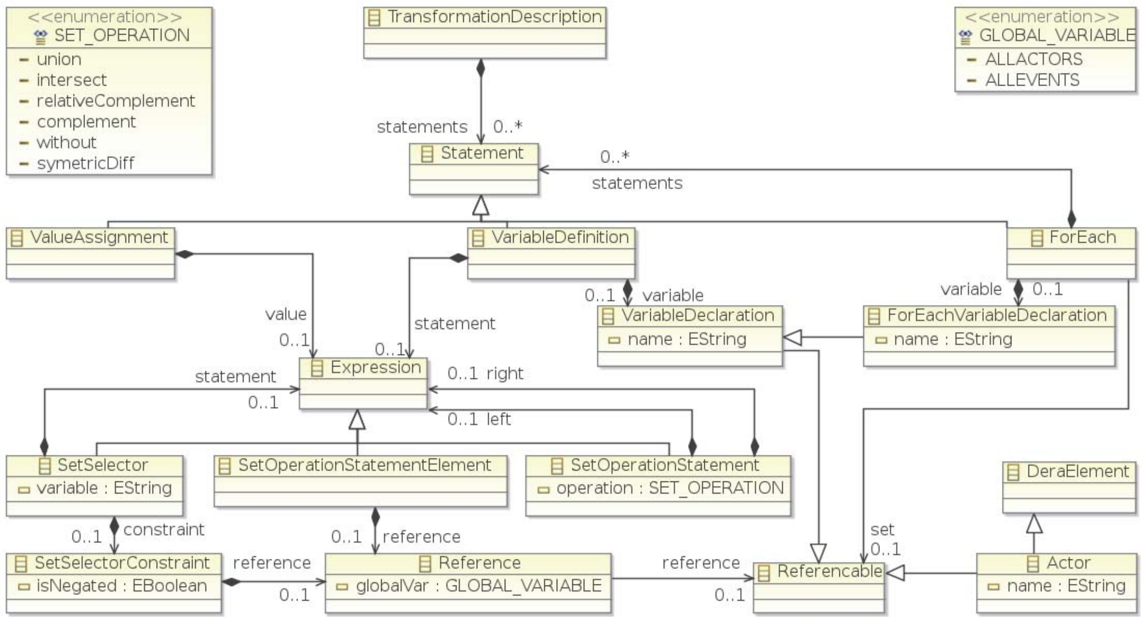

Fig. 12. Outline of the Change Pattern Description DSL Model - Transformation Description

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.

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 [17]. Unfortunately, they often assume explicit invocations between elements, and therefore, are not readily applicable for event-based architectures. The only technique to extract implicit invocation information from an event-based architecture, proposed by Murphy et al. [18], is Lexical Source Model Extraction (LSME). However, the results are imprecise and incomplete. Another approach to analyze event-based architecture is proposed by Jayaram et al. [19], which aims at extracting type information and dependencies at compile time, based on EventJava [20]. Analysis at runtime, or after changes applied, are not supported. Program slicing techniques [21], [22] 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 architectures are developed using the DERA framework.

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. [23]. 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 architectures are less strict than Helios and easy to be satisfied by existing event-based architectures. Moreover, we introduce appropriate abstractions and techniques for supporting the developers in analyzing and performing different types of changes on an event-based architecture.

Message oriented middleware can be used to develop event-based architectures. Components communicate with each other via ambiguous interfaces, which accept a single, abstract message type which holds the specific data to be interpreted by the receiving component. Garcia et al. [24] proposes Eos to identify message types and component dependencies. Both, Helios and Eos rely on lexical source model extraction. In contrast, our work can extract all needed information either from a model or from meta data of running instances.

Since all of the existing approaches for impact analysis for event-based architectures 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.

VI. Lessons Learned

We claimed in our previous work that modeling changes on different levels of abstraction [12] will support developers in changing event-based architectures and enables conflict and impact analysis of changes. In this work, we presented models for describing change patterns and changes for event-based architectures by providing domain specific languages for each level of abstraction. We also showed the feasibility of providing DSL editors and generators for each level of abstraction.

Patterns support understandability of solutions for problems in a specific context. A catalog of patterns will grow with the knowledge of the developers using these patterns. New patterns will be discovered and variants identified. We illustrated this kind of pattern evolution using the example of inserting a new actor into a DERA application in Section IV-B, leading to two variants **ParallelInsert** and **SerialInsert** with different semantics. Therefore, one conclusion of our work is that a static catalog of patterns will not suffice. Our Change Pattern Description DSL still does not meet all requirements to express variants. For example, the affinity between related patterns can currently not be expressed.

We already proposed some change patterns as shown in Table II, for instance in [11]. These patterns only address a change related to exactly one actor. For instance, **Delete** will cause one actor to be removed from an execution domain. As we deal with sets of actors, this restriction prevents us from describing effective change patterns. One reason is that we should support describing the context of a changing actor. For instance, deleting an actor needs to specify one ore many predecessors and successors so that the input and output ports of these predecessors and successors can be adapted. We are currently working on a catalog describing the semantics of the change patterns, supporting sets of actors rather than a single actor, also considering predecessors and successors. The patterns used in this paper already support the usage of actor sets.

We provide a model-based solution for defining a catalog of change patterns for event-based architectures, which explicitly defines the semantics of a certain change by defining set-based transformation operations. We currently support only the basic operations and control structures (e.g., for each) which are needed to describe our current set of change patterns. Over time, the set of operations and control structures may have to be extended.

We can use our models proposed in this paper to run impact and conflict analysis, as described in [12], [25]. The models can also be used to run some optimization. For instance, the sequence of primitives to be executed can definitively be optimized, as you can see in Figure 7 where some lines may have no effect (e.g., applying an empty set of event types) and some lines are redundant. Also, we have to improve our toolset, so that the results from analysis can be used at all levels of abstraction. In our next steps we will use our model-based approach to bring the computed information from the change primitives level to the change pattern level, so that the developers will be able to estimate the impact of a change by interpreting the results of the impact and conflict analysis directly shown in the change pattern editor.

VII. Conclusion

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 this paper by introducing model-based domain specific languages to express changes on different levels of abstraction. On the highest level, developers can describe patterns of change, which can be transformed to a Change Pattern DSL. Instances of change patterns can be transformed to sequences of change primitives at the lowest level of abstraction. The change primitives can be applied to DERA, our implementation of a distributed event-based architecture. As only the very basic concepts of the DERA Meta Model are used, our results can be generalized to other implementations of event-driven architectures rather easily.

As a result of our approach, the architect of a distributed event-driven architecture can 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.

As future work we plan to study patterns for more specific situations in event-driven architectures that go beyond the simple change patterns. We also want to empirical study the impact of event-driven architectures on designing and evolving distributed systems.

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. New York, NY, USA: ACM, 2010, pp. 185–194.

[5] 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.

[6] M. Reichert and B. Weber, *Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies*. Springer, 2012.

[7] W. M. P. van der Aalst, M. Pesic, and H. Schonenberg, "Declarative workflows: Balancing between flexibility and support," *Computer Science - Research and Development*, vol. 23, no. 2, pp. 99–113, Mar. 2009.

[8] H. Schonenberg, R. Mans, and N. Russell, "Process flexibility: A survey of contemporary approaches," in *The 4th Int'l Workshop CIAO! and 4th Int'l Workshop EOMAS*. Springer, 2008, pp. 16–30.

[9] 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.

[10] ——, "Event actors based approach for supporting analysis and verification of event-driven architectures," in *17th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE Computer Society Press, September 2013, pp. 217–226.

[11] 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.

[12] S. Tragatschnig and U. Zdun, "Modeling Change Patterns for Impact and Conflict Analysis in Event-Driven Architectures," in *24th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, International Track on Adaptive and Reconfigurable Service-oriented and component-based Applications and Architectures*. Larnaca, Cyprus: IEEE, June 2015.

[13] 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-Verlag, 2008, pp. 279–293.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] P. Eugster and K. Jayaram, "Eventjava: An extension of java for event correlation," in *ECOOP 2009 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, S. Drossopoulou, Ed., vol. 5653. Springer Berlin / Heidelberg, 2009, pp. 570–594.

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

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

[23] 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)*. New York, NY, USA: ACM, 2012, pp. 241–251.

[24] J. Garcia, D. Popescu, G. Safi, W. G. Halfond, and N. Medvidovic, "Identifying message flow in distributed event-based systems," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Saint Petersburg, Russian Federation: ACM, 2013, pp. 367–377.

[25] 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.