Supporting the Evolution of Event-Driven Service-Oriented Architectures using Change Patterns

Simon Tragatschnig, Srdjan Stevanetic and Uwe Zdun
Software Architecture Research Group
Faculty of Computer Science
University of Vienna
Vienna, Austria
{firstname.lastname}@univie.ac.at

Abstract

Context: The components of an event-driven service-oriented architecture (EDSOA) are composed in a highly decoupled way, facilitating high flexibility, scalability and concurrency in SOA systems. Evolving an EDSOA is challenging because the absence of explicit dependencies among constituent components makes understanding and analysing the overall system composition difficult. The evolution of EDSOAs typically happens by performing a series of primitive changes—which can be described formally as change primitives.

Objective: In this article, we present our change pattern based approach for managing the EDSOA evolution as a novel design method supporting EDSOA evolution. The change patterns operate on a higher abstraction level than change primitives.

Method: To evaluate our approach, we have compared both time and correctness of changes in a controlled experiment comparing the understanding and performing of changes in EDSOAs. The experiment has been conducted with 90 students of the Software Architecture course at the University of Vienna. We compare the efficiency of 3 sets of change operations for modifying a given system architecture to obtain a desired architecture: a minimal set of 3 change patterns, an extended set of 5 change patterns, and a minimal set of 4 change primitives.

Results: Our results show that change patterns based evolution requires significantly less time to capture a similar level of correctness as the evolution based on change primitives, presuming that a certain level of transformation complexity is required. Furthermore, we did not observe a significant difference in the correctness level nor in the time required to perform the changes using an extended pattern set compared to a minimal set of patterns.

Conclusions: We clearly show the feasibility of our approach by developing a design method and tool support using a model-driven tool chain consisting of 3 domain-specific languages and empirically evaluating the approach in a controlled experiment.

1 Introduction

In recent years, distributed event-driven architectures have become widespread in their use in several domains such as real-time control systems, stock market and fast trading, network intrusion detection, sensor networks, healthcare monitoring, mobile and wearable computing (see e.g. [20, 26]). A main reason is that event-driven architectures provide solutions for developing distributed systems that facilitate high scalability, flexibility, and concurrency [20]. An event-driven architecture typically comprises a number of computational or data handling elements (i.e., components, actors) that communicate with each other by sending and receiving events [20]. Each component may independently perform a particular task, for instance, access a data storage, dispense cash from a credit card, or interact with users. Nowadays the components or actors in event-driven architectures are typically services, leading to a combination of event-driven and service-oriented architecture concepts, coined with the term event-driven service-oriented architecture (EDSOA) [13, 21]. More precisely, we use the term to describe event-driven architectures that are used to realize flexible service communication and orchestrations [13, 23, 50].

The communication style used in EDSOAs is based on implicit invocations performed by publishing an event (or message) to an event channel (also called event bus or message broker) instead of explicit invocations where one component is directly called via a reference [26]. As the exchange of events among the components is performed through the event channel, every component is in principle unaware of the others. This way a high degree of flexibility in the system is supported.

For example, the execution order of components can be changed (e.g., re-routing or adding some components) or any component can be replaced (e.g., with a bug-fixed or upgraded version) whilst the system is running. Additionally, EDSOAs support high scalability since the loosely coupled components can be executed concurrently and easily placed on different hardware nodes or virtual machines. However, the additional wanted degrees of flexibility, scalability, and concurrency through loose coupling might also increase the difficulty and uncertainty in understanding, maintaining and evolving these systems [8].

As requirements on software systems evolve over time, they have to be constantly maintained and changed [18]. More than one quarter of coding time is spent on implementing changes and investigating their impact [16]. By analyzing evolution of software systems, Weber et al. identify a set of change patterns that recur in many of existing software systems [45]. These patterns are specific for process-aware information systems (PAIS) where the execution of the software system is bound to a process schema, a prescribed rigid description of the behavior flow, and therefore, mostly can not be changed during runtime or just slightly deviated from the initial schema [30, 34, 42]. As a result, these

approaches are not easily applicable for EDSOAs where components are highly decoupled and the dependencies between components are subject to change at any time, even during the execution of the systems. Nevertheless, these patterns provide a basis for describing changes of the behavior in any information system.

To deal with the complexity and the large degree of flexibility of EDSOAs, a set of change operations that enable modifications in the system at different abstraction levels is proposed in our approach. Those change operations include low-level primitives (change primitives) for encapsulating the primitive change actions, such as adding or removing an actor or event, and high-level patterns (change patterns), that encompass a number of change primitives. An example of a change pattern is replacing an actor that represents a service call. This replacement pattern encompasses 'removing and adding an actor' primitives as well as 'removing and adding events' primitives. The provided set of patterns significantly extend the change patterns that are frequently occurring in most information systems [45] in order to deal with the specifics of EDSOAs. Hence our first research question addressed in this article is:

Research Question 1 (RQ1): Can the concept of change patterns (as defined for information systems by Weber et al. [45]) be used as a foundation for a design method for the evolution of EDSOAs, reflecting the specific changes required in EDSOAs?

In this article, we also investigate how the previously mentioned change operations (change patterns and change primitives) influence the process of performing changes or evolving an EDSOA. More precisely, we hypothesize a positive effect on the efficiency of performing changes in an EDSOA model using the change patterns compared to change primitive based changes. Hence, the second research question we studied was:

Research Question 2 (RQ2): If RQ1 can be answered positively, is there a significant positive influence on the efficiency of performing changes in an EDSOA model using the change patterns compared to change primitive based changes?

To study this research question, we conducted a controlled experiment where we compared the efficiency of change patterns and change primitives for modifying a given system architecture in order to obtain a desired one. The participants of our study were 90 students of the Software Architecture lecture at the University of Vienna. They were divided into 3 groups, and each of them was asked to modify a given system architecture (called the source architecture) using a given set of change operations in order to obtain a desired architecture (called the target architecture). For all groups the same 4 source/-target pairs of EDSOA architecture models were given. The first group was provided with a set of 3 change patterns which also represents a minimal set of change patterns needed to perform any change in the system. The second group was provided with two additional patterns, to enable us to study to which extent additional higher-level abstractions help. Finally, the third group was provided with a set of 4 change primitives,

which also represents a minimal set of primitives required to perform any change in the system. Our results show that, for the most complex model studied and the case where all models are considered together (the results for all studied models are summed up), the group with change primitives required significantly more time to reach a similar correctness level of pursued changes compared to the groups with change patterns. The obtained results provide empirical evidence that change patterns based evolution is generally more efficient than change primitives based evolution of EDSOAs, presuming that a certain level of transformation complexity is required. Moreover, the subjects used two additional patterns in the extended pattern set only to a limited extent and had problems with their correct application. No significant difference in the correctness level of pursued changes nor in the time required to capture those changes using the extended pattern set compared to the minimal set of change patterns was observed.

The major contribution of this article is the empirical study on efficiency of performing changes in EDSOAs, to find answers to our research questions RQ1 and RQ2.

In our previous work [1, 37–39], we investigate and adapt change patterns in the context of event-based architectures dealing with the lack of prescribed execution descriptions and the potentialy aribrarily changed relationships between constituent elements of a system. In order to deal with the complexity and the large degree of flexibility of event-based architectures, in this article we combine our prior works into a novel design method for the evolution of EDSOAs. As another novel major contribution, we present an empirical study in which we evaluated our approach with 90 participants.

This article is organized as follows: In Section 2, we discuss the related work, and in Section 3, we describe the required background on EDSOAs and change operations in their context. We then describe our change pattern Based design method for supporting EDSOA evolution in Section 4. Next, in Section 5 we discuss our empirical study on the efficiency of performing changes in EDSOAs using our pattern based approach. Finally in Section 6, we summarize our main contributions.

2 Related Work

2.1 Related Works on Change Patterns

Starting from the seminal work on the evolution of software systems by Lehman [17], several techniques for supporting different types of system evolutions have been investigated in different application domains [2]. One of the important works presented by Weber et al. [31] identified a large set of change patterns that are frequently occurring in the most of today's process-aware information systems (PAIS). The change patterns observed by Weber et al. targeted PAISs in which the execution order of the elements

are prescribed at design time and unchanged or slightly deviated from the prescribed descriptions at runtime, therefore, not readily applicable for event-based systems where components are highly decoupled from each other. However, these patterns can be used as a basis for defining the corresponding change patterns for event-based systems (see e.g. [37–39]), since the structure of PAISs (i.e. process instances and their connections) can in principle be mapped to the structure of EDSOAs (i.e. event-based components and their connections). But, the specificities of EDSOAs in terms of e.g. the events that the EDSOA components send and/or receive or the execution domains need to be taken into account additionally. For instance, in PAISs the change primitives only deal with adding or deleting a node or an edge, while in EDSOAs additional primitives like e.g. replace an event of the component's input and/or output port or remove a set of events from the port (see [37]) need to be considered.

Because of the loose coupled nature of EDSOAs, different variants of change patterns known from PAIS with different semantics may exist. For instance, an *Insert* in PAIS must be distinguished from *ParallelInsert* and *SerialInsert*, as described in [1]. Therefore, we have implemented a subset of patterns presented by Weber et al. to meet the requirements for EDSOAs (see Section 4.2).

The broader variety of possible changes has led us to an extensible solution, rather than a smaller, more fixed set of change patterns such as those for information systems by Weber et al. [45]. For instance, inserting a new element in EDSOAs will lead to different variants of the **Insert** pattern by Weber et al. with different semantics. In [1] we identified the variants **ParallelInsert** and **SerialInsert**.

2.2 Related Work on Event-driven Systems

Due to the loosely coupled nature of the participants, event-based architectures have been investigated and leveraged in many large-scale distributed software systems today [8, 19, 20]. The advantages of event-driven communication styles have been extensively studied in numerous approaches in different areas [10, 23, 36].

To the best of our knowledge, there are no prior empirical studies on the understanding of the event-driven architectures at runtime. Some studies have been conducted on the comprehension of general software models [5, 22, 27] but most of them focus on the syntactical structures of the models under consideration. Therefore, they have not considered the understanding of the dynamic interplay of models at runtime nor investigate the factors that influence the aforementioned understanding.

In our previous work [37–39] we have proposed a set of change operations that enable modifications of event-driven architectures at different abstraction levels. Particularly, change primitives are proposed to capture the low-level modification actions and therefore

can be efficiently leveraged by technical experts. Additionally, on top of the primitive actions a number of change patterns that encapsulate essential change actions that recur in many software systems is proposed. Recurring patterns are not readily applicable for event-based systems where components are highly decoupled and the dependencies between them can change at any time, even during the system execution. Therefore, as mentioned above, those patterns were used just as a basis for developing an adapted set of patterns that are specific for event-based software systems. In this study we provide an empirical evaluation of the usefulness of both change primitives and patterns for performing changes in the system.

One of the existing software development approaches for building distributed event-based systems (DEBS) is CEP (Complex-Event Processing) [8]. CEP is defined as a set of tools and techniques for analyzing and controlling a complex series of interrelated events [19]. Another approach is using a DEBS middleware, such as a messaging middleware or a publish/subscribe system, to base distributed communication on events (see e.g. [11]). With regard to our approach, CEP and DEBS focus more on detecting, processing and responding to events as message streams, while we focus more on architectural aspects of modelling the event-based systems and their evolution. With regard to the specification and verification of event-driven architectures, a number of approaches exist that use temporal logic [8], trace semantics [8], algebraic semantics [7], and similar formalisms.

2.3 Related Work on EDSOAs

There exist several studies that discuss the advantages of an integration between service-oriented and event-based architectures (SOA and EDA). Niblett and Graham [21] have illustrated how the combination of EDA and SOA into a single EDSOA infrastructure brings many advantages, because it is quite common for a single service to combine both request/response and event-oriented message exchanges. The research of Yuan and Lu [50] shows an EDSOA based on a concept of value-centric processing and the event-based communication style. Event-based communication offers a service provider to have more information about their clients by creating client profiles. Juric [13] discussed how SOA can be extended with EDA concepts. In that approach services act as event producers and event consumers and at the same time preserve the interfaces and their operations. The approach also enables event-based service orchestrations in business processes.

In the field of coordinated service integrations there exist several approaches that emphasize the integration between service oriented and event-driven architectures (SOA and EDA). These approaches share many of the same characteristics such as modularity, loose-couplings, and flexibility [13, 23].

The aforementioned approaches mainly target the publish/subscribe messaging pattern. None of those approaches aims at introducing appropriate representations and formalisms

of event-based systems that are close to the developers' perception. High-level concepts and representations for modeling and developing event-based systems have been presented in our previous work [41]. In this work we utilized those high-level concepts to investigate the understandability of EDSOAs.

2.4 Related Work on Process-Driven Systems

In the area of workflow patterns different authors provided a thorough examination of the various perspectives that need to be supported by a process specification language and process modelling tools (see for example [32, 33]). However, none of these approaches provides empirical evidence in which way such patterns affect process modelling. Thom et. al [35] reported on activity patterns for designing process models. They performed an empirical study using 214 process models from different application domains to examine the frequency with which different activity patterns occur in real process models. The analysis was accomplished in order to verify whether candidate process fragments may be considered as patterns with high probability for reuse. The results showed that the detected patterns are well suited for defining both business processes and workflows from a variety of application domains.

The creation of process models based on change primitives has recently received considerable attention resulting in research on the process of process modelling (PPM) [3, 25]. This research focuses on the formalization phase, i.e., the interactions of the process modeller with the modelling environment. As mentioned above, Weber et al. [31, 45] identified a large set of change patterns that are frequently occurring in and supported by the most of today's process-aware information systems. In the context of process models creation, several change patterns have been investigated [6]. Other articles provide empirical insights into the usage of change patterns in the process of process modelling (PPM) [43, 46]. For example, some of the results indicated that an extended change pattern set puts an additional burden on modellers who perceive them as more difficult to use. Therefore, an expected increased problem solving efficiency was not achieved. Contrary to the mentioned studies, in our study we compare the efficiency of change primitives and change patterns during performing changes in the system. In our study, we also studied an extended pattern set but contrary to the given studies that used move patterns that enable transformations on composite system fragments we used basic patterns that deal with a single actor in the system.

To a certain extent studies on flexible process-aware systems are related to our work, too, but we are not aware of a similar empirical study in the area of flexible process-aware systems. Several approaches try to relax the rigid structures of process descriptions to enable a certain degree of flexibility of process execution [12, 29]. Event-based systems provide a high flexibility for runtime changes, since there exist no explicit relationships

among the actors.

3 Background on Event-Driven Service-Oriented Architectures

To support performing specific changes in EDSOAs as well as understanding their impacts, we assume that each service exposes an event-based interface that specifies a set of events that the service can receive (the *input events*) and a set of events that the service will emit (*the output events*). The input events trigger the execution of a certain service so the service is considered as "consumer". In turn, a service can also emit one or many output events after its execution finishes and therefore is called "producer".

The service input or output events can only be observed at a certain point in time since a service's interface can be changed whilst the system is running. New events can be added to the interface or some events can be removed from it. Using the services' exposed interfaces the dependencies between the services can be extracted at any time without examining the source code. The concept of exposed interfaces fully complies with the case of third-party services that are usually provided as black-boxes with documented interfaces.

In general, the above mentioned concepts can be satisfied by most existing event-based systems [20] when used for service orchestration. For demonstration purpose, in our previous work we built upon the DERA framework [40] that provides fundamental concepts for modelling and developing EDSOAs and complies with the concepts and assumptions mentioned above.

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 article, we focus on the core concepts of DERA only. A detailed description of DERA can be found in [40]. In DERA, a service 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" [20] (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 [40], differing slightly in its behavior. For the sake of simplicity we do not explain them in

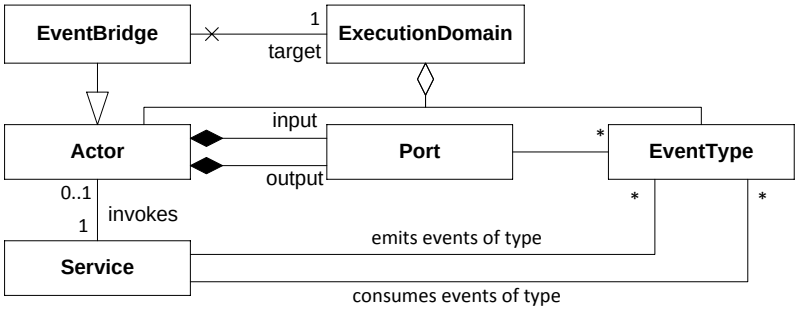

Figure 1: Simplified Excerpt of the DERA Meta Model (only the necessary elements needed for this article)

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 [40].

In a EDSOA context, the DERA runtime uses actors to invoke services which—via the input and output ports of the actors—emit and consume events of the event types specified by those ports. The actor is a one to one representation of the service in the DERA runtime or an internal actor that is not connected to a Web service.

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 turn, A is a *predecessor* of B. This means, that a *successor* is executed after the execution of its *predecessor* has finished.

Based on these basic notions of event-based systems, in our previous work we have proposed a set of change operations at different abstraction levels that enable performing different modifications of EDSOAs (see [37–39]). On the lower abstraction level, change primitives are used to express fine granular changes, while, on the next higher level, change patterns that encompass a number of change primitives are used to express more complex changes of the system, such as moving an actor. The given change patterns are derived from similar patterns that are frequently occurring and supported in most of today's information systems, according to the survey presented by Weber et al. [45]. However, recurring patterns are not readily applicable for EDSOAs where services are highly decoupled and the dependencies between them can change at any time, even during the system execution. Therefore, those patterns were used just as a basis for developing an adapted set of patterns that are specific for EDSOAs (see [37] for more details).

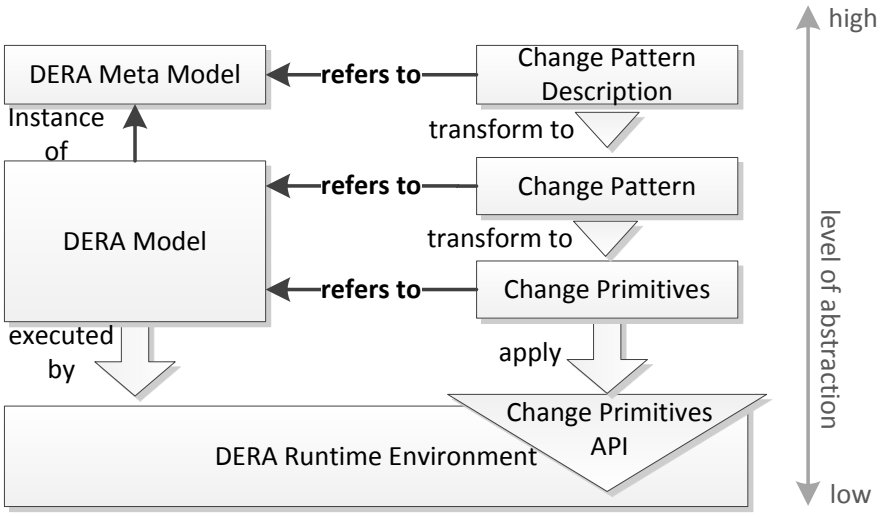

Figure 2: Overview of Change Operations in Relation to DERA Models

4 Change Pattern Based Design Method for Supporting EDSOA Evolution

Maintaining EDSOAs 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 source code. Assisting techniques for analyzing the impacts of certain changes are missing, hindering the implementation of changes in EDSOAs. Specifying a design method supporting the evolution of EDSOAs allows developers to focus on their concepts, like events and event emitting or consuming components.

Change patterns and change primitives express changes on different levels of abstraction to deal with the complexity and the large degree of flexibility of EDSOAs, as shown in Figure 2. 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 change patterns.

We addressed the different levels of abstraction for change primitives and change patterns, as well as a modeling approach using domain specific languages in our previous work [1, 37–39]. The following subsections will give an overview of our existing work.

	Change Primitives Specification	Description
INSERT	*ExecutionDomain* +=actors	Add the actor a to the execution domain
DELETE	*ExecutionDomain* -=actors	Remove the actor a from the execution domain
SET DOMAIN	*Actor->domain* =domain	Set the execution domain d for actor a
ADD	*Actor:Port* +=events	Add a set of *events* to port p
REMOVE	*Actor:Port* -=events	Remove a set of *events* from port p
REPLACE ALL	*Actor:Port* =events	Replace all events of port p with another set of *events*

Table 1: Outline of change primitives

4.1 Change Primitives

We use low-level primitives, called change primitives introduced in [37], for encapsulating the basic change actions for populating and modifying EDSOAs, such as 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. Table 1 shows the change primitives and summarizes their effects. In our model-driven prototype implementation (see Section 4.3), we provide generators to transform instances of modeled change primitives to executable code, which can be performed by DERA. Based on these primitives, in the following Section 4.2 we present change patterns for EDSOAs with which the software engineers can easier describe and apply desired changes at a higher level of abstraction.

The change primitives encapsulate the primitive change actions for populating and modifying EDSOAs. In our controlled experiment, we use a subset of these low level change operations which represents a minimal set of change primitives: Add, Insert, Delete, and Remove. This subset has been selected, as they are those primitives used in the realization of the change patterns used in the other groups of the experiment.

4.2 Change Patterns

Change patterns for EDSOAs support software engineers to describe and apply desired changes at a higher level of abstraction than the primitives. 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 [45]. Table 2 gives an overview of our realized change patterns for EDSOAs.

A change pattern basically expresses that a set of actors should change its position within a DERA application, related to other implicitly dependent actors with matching interfaces. A change pattern consists of various statements, which describe a change. A change defines a set of source actors, defining 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. To illustrate the patterns, Figures 3 and 4 illustrate the realized insert

patterns, both with an example to show the difference. For the formal semantics of the patterns please refer to [37].

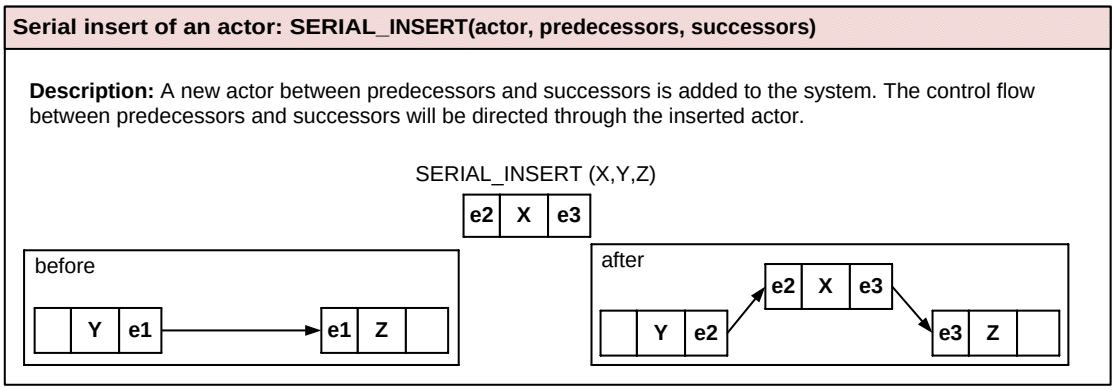

Figure 3: Serial Insert Change Pattern

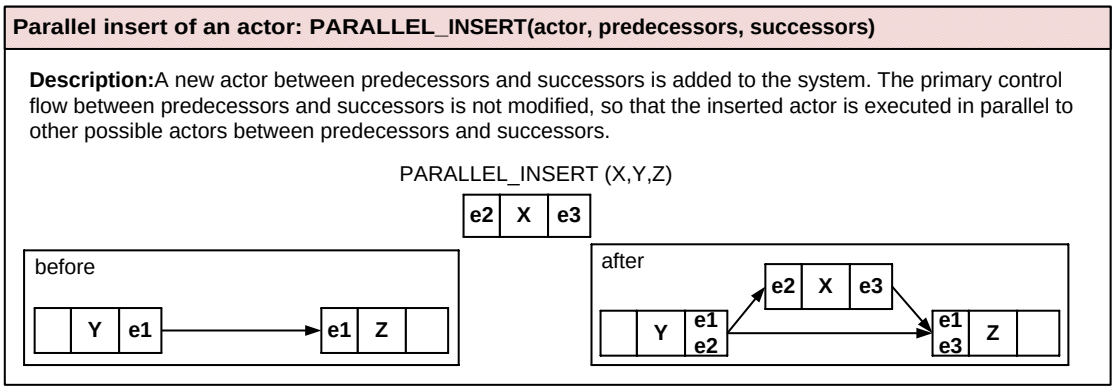

Figure 4: Parallel Insert Change Pattern

We developed a model-based change pattern prototype (see Section 4.3) which is designed to support developers expressing changes without detailed knowledge of the DERA framework. The realized change patterns are listed in Table 2, coming with generators to transform change pattern instances into a set of change primitives explained in Section 4.1.

4.3 Model-driven Tool Support

Our model-based tools and their transformations are developed with XText[1] and Xtend[2] Framework for Eclipse, which gives us the benefits of code completion, text suggestions and validation within the Eclipse IDE. We developed two integrated model-based domain-specific languages (DSL) for change patterns and change primitives that are able

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

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)	Moves the actor x in a way that the actor y becomes predecessor and the actor z becomes 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 2: An overview of change patterns

to express changes on different level of abstraction to deal with the complexity and the large degree of flexibility of EDSOAs.

On the lower abstraction level, change primitives are used to express fine granular changes on the modeled DERA application. The language to express change primitives is explained in Section 4.3.1.

Change patterns are defined at a higher abstraction level, as described in Section 4.3.2. Using model transformations they are transformed into a set of change primitives.

Both DSLs are supported through Xtext-based Eclipse editor plug-ins and use Xtend for model transformations and code generation. The DERA code generator generates code for the DERA applications which consists of one or many DERA runtimes, orchestrating web services. The respective components of our tool chain described in this section are shown on the right hand side of Figure 5.

4.3.1 DSL for Change Primitives

At the lowest level of abstraction, change primitives express very basic change operations. Our domain specific language to express change primitives was introduced in [1], which is based on the core DERA concepts described in [40]. Basically, the primitives can describe an expression to add, set or remove DERA elements. A DERA element can be an ExecutionDomain, an Actor with ActorPorts, or an Event. A structural overview of the Change Primitives DSL is shown in Figure 6.

By referencing a DERA model instance, it is possible to reference instances of specific DERA elements. An example instance of the Change Primitives DSL is shown in Figure 7. It is the specific result of the Change Pattern DSL example explained in Section 4.3.2. In the Change Primitives DSL example, a set of primitives are shown to change the DERA application *orderApp*. The lines with the marker *(1)* and *(2)* removes a set

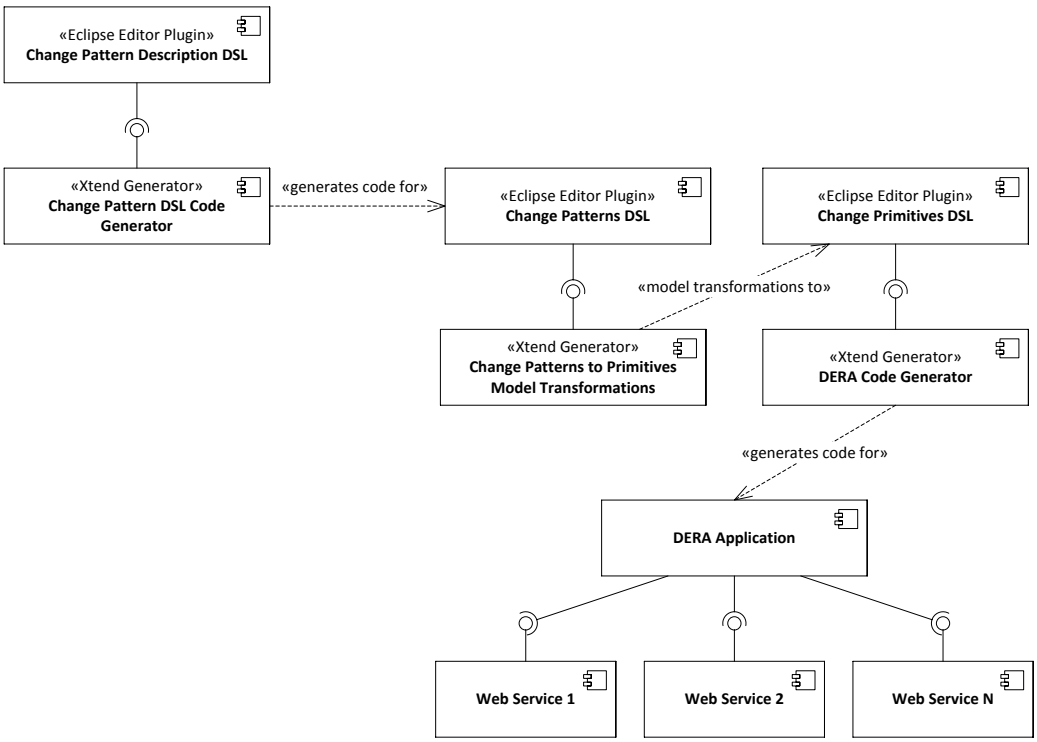

Figure 5: Model-driven Tools and their Relation to the DERA application and Orchestrated Services

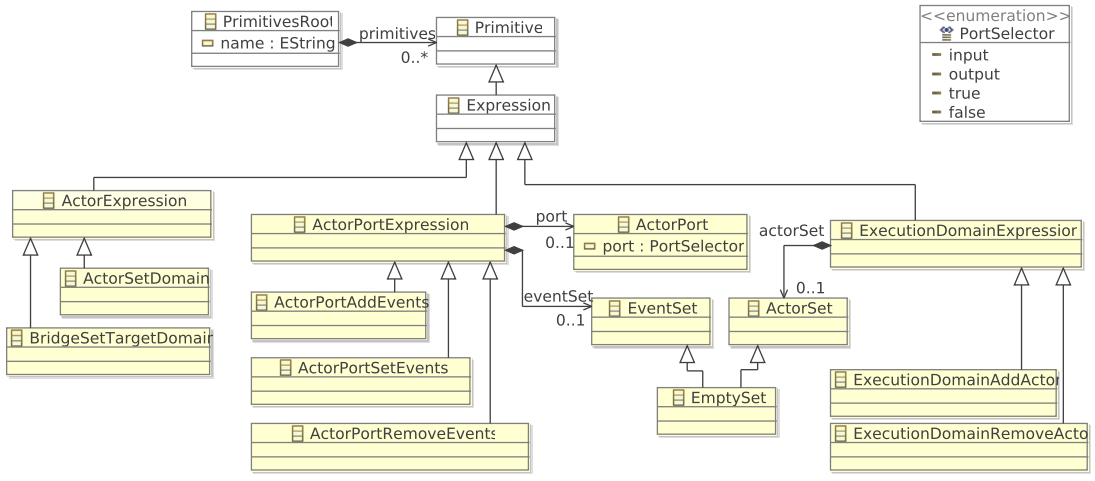

Figure 6: Model of the Change Primitive DSL

```
primitives IllustrativeExample
{
  //prevent loops
  orderApp.ExternalService:input  -= {}; // (1)
  orderApp.ExternalService:output -= {}; // (2)
  //insert elements
  orderApp.OrderTranslator:input += orderApp.orderV2; // (3)
  orderApp.OrderReceiver:input   += orderApp.orderV1; // (4)
}
```

Figure 7: Change primitives for the SerialInsert

of events from the *input* and *output* port of the actor *ExternalService*. In this specific example, the set is empty, but may contain a set of events for more complex DERA applications. The line with the marker *(3)* adds the event *orderV2* to the input port of the actor *OrderTranslator*. The line with the marker *(4)* adds the event *orderV1* to the input port of the actor *OrderReceiver*

4.3.2 DSLs for Change Patterns

At a abstraction level above the change primitives, change patterns can be described to express a change. As the Change Pattern DSL is referencing a DERA model instance, specific DERA Elements as actors or events can be referenced. The Change Pattern DSL was introduced in [1]. Basically, the Change Pattern DSL can express a specific change of a DERA application, by describing the change of an actor's position related to its predecessors and successors after the change is applied. A simplified structural overview of the Change Pattern DSL is shown in Figure 8. To describe a change pattern, sets of actors can be selected which may be affected by the change. The *SourceSelector* references the actors which should change its position. The *TargetSelector* describes the set of preceding and succeeding actors, when the change is applied. To address more complex relations between actors, *Constraints* can be described to express, which events should not be affected by the change.

An example of a **SerialInsert** is shown in Figure 9: The actor *OrderTranslator* (the *SourceSelector*) must be inserted in a way that the actors of the *TargetSelector* will be its predecessor (*ExternalService*) and successor (*ExternalService*).

Based on a Change Pattern DSL instance, the change primitives are derived to enact the change.

4.3.3 Change Pattern Description DSL for Supporting Change Pattern Definition

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

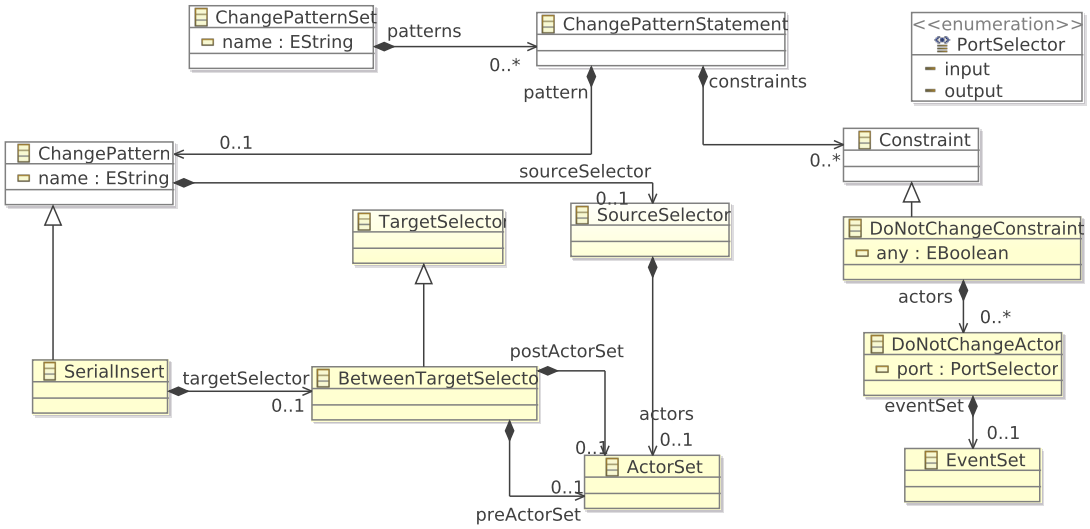

Figure 8: Simplified model of the Change Pattern DSL

```
Change Set IllustrativeExample {
    SerialInsert orderApp.OrderTranslator
    between orderApp.ExternalService
    and orderApp.OrderReceiver.
}
```

Figure 9: SerialInsert change pattern

change pattern or pattern variants. Therefore, we decided to develop a DSL to describe change patterns and their impact, as introduced in [1]. That is, from textual models of change pattern descriptions, we can generate the code for the Change Pattern DSL described in Section 4.2. This way, the Change Pattern DSL's definition is highly extensible and easy to change.

Consider the pattern **Insert** introduced above to illustrate the possible 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 actor is inserted. A detailed description of both variants can be found in [37].

As a second example of pattern variation let us consider the pattern **Move** and its pattern variants: **Move** describes how to change position of an actor within the implicit execution path. One possibility is to just add the new dependency information to the ports of the actor being moved, without changing any other actor (non-disruptive move). This may cause side effects like loops or parallel execution paths. Another variant of the pattern **Move** may isolate the actor being moved completely from it's implicit successors and predecessors by changing their ports before the actor is inserted on it's new position

```
Catalog variants.insert.patterncatalog
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
     predecessor:in = predecessor:in
       without (toBeInserted:out intersect predecessor:in)
     predecessor:out = predecessor:out
       without (predecessor:out intersect successor:in)
     // insert
     toBeInserted:in = toBeInserted:in union predecessor:out
     successor:in = toBeInserted:out union
        (successor:in without (predecessor:out intersect successor:in))
}
```

Figure 10: Change pattern description of **SerialInsert**

(disruptive move). This may prevent loops and parallel execution paths, but may cause other side effects like dangling or dead actors.

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 operations 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 Eclipse. The additional Change Pattern Description DSL is shown on the left hand side of Figure 5.

Figure 10 shows the definition of a **SerialInsert** using Change Pattern Description DSL. The descriptions contains a name, a short textual description, simple syntax definition and the pattern's semantics expressed by set-based transformation rules.

4.4 Discussion

In this section, we have illustrated the challenges of modeling change patterns for EDSOAs and especially the significantly extended scope compared to the change patterns for information systems by et al. [45]. This extended scope is mainly due to the complexity and the large degree of flexibility of EDSOAs. However, as model-driven support for a number of variants of basic change patterns based on established change primitives is possible, as well as extensible tool support to deal with the wide variety of possible pattern variants in EDSOA evolution, we can conclude that RQ1 can be positively answered: Indeed, the concept of change patterns (as defined for information systems by

Weber et al. [45]) can be used as a foundation for a design method for the evolution of EDSOAs, reflecting the specific changes that EDSOAs require.

The broader variety of possible changes has led us to an extensible solution, rather than a smaller, more fixed set of change patterns such as those for information systems by Weber et al. [45]. For instance, inserting a new element in EDSOAs will lead to different variants and semantics of the **Insert** pattern identified by Weber et al. In [1] we identified these variants as **ParallelInsert** and **SerialInsert**.

The need of expressing variants of patterns with different semantics led us to the design of the Change Pattern Description DSL described in the previous section.

5 Empirical Study on Efficiency of Performing Changes in EDSOAs

5.1 Empirical Study Description

As RQ1 could be positively answered, we have further studied RQ2 in a controlled experiment. For the study design, we have followed the experimental process guidelines proposed by Kitchenham et al. [14] and Wohlin et al. [48]. The former was primarily used in the planning phase of the study while the later was used for the analysis and the interpretation of the results.

5.1.1 Goal, hypotheses, and variables

As mentioned before, this study examines how the change operations for EDSOAs defined at different levels of abstraction (i.e., change primitives and change patterns) affect the process of performing changes on those architectures. Namely, we compared the performances among 3 groups of participants each of them was provided with a different set of change operations. The first group was provided with a set of 3 change patterns (Serial Insert, Parallel Insert, and Delete) that represents a minimal set of change patterns for performing any change in the system. The second group was provided with two additional change patterns (Reroute and Replace) while the third group had to deal with a set of 4 change primitives—Add, Insert, Delete, and Remove—which also represents a minimal set of primitives for performing any change in the system and are those primitives needed to perform the changes described in the 5 selected change patterns.

In our study, we focus on patterns that operate on one element (i.e., actor) in the system such as adding a new actor, deleting an actor, or replacing an actor with a new one. As mentioned above, those patterns consist of 3 basic patterns (Serial Insert, Parallel Insert, and Delete) that represent a minimal set of patterns for capturing any change

Description	Scale type	Unit	Range
Time (Time)	Ratio	Minute	Natural numbers incl. 0
Correctness (CoCC)	Ratio	Change	Natural numbers incl. 0

Table 3: Dependent Variables

in the system and two additional patterns (Reroute and Replace, see Table 2) that can capture more complex changes requiring a combination of basic change patterns. Under the assumption that the expression power of change patterns to capture several changes significantly predominates over the difficulty to foresee the impact of introducing them, we expect that the change patterns (both the basic and the extended ones) provide more efficient way of capturing changes in the system than the change primitives. Similarly, we also expect that the extended patterns set is more efficient than the basic patterns set. The efficiency of performing changes is captured through 2 dependent variables in our study, the correctness of the captured changes (CoCC) and the time required to perform required changes (Time). Beside these 2 dependent variables, we introduced 7 independent variables concerning the participants' experience (UML modelling, programming, and programming of distributed systems) and group affiliation (3 different groups of participants).

Based on the discussion above, we can break down our research question RQ2 into the following hypotheses to be tested in our study:

Hypothesis $\mathbf{H_1}$: The usage of both a minimal and an extended change patterns set significantly increases the correctness of the captured changes in EDSOAs compared to the usage of a minimal set of change primitives.

Hypothesis $\mathbf{H_2}$: The usage of both a minimal and an extended change patterns set significantly decreases the time required to perform required changes in EDSOAs compared to the usage of a minimal set of change primitives.

Hypothesis $\mathbf{H_3}$: The usage of an extended set of change patterns significantly increases the correctness of the captured changes in EDSOAs compared to the usage of a minimal change patterns set.

Hypothesis $\mathbf{H_4}$: The usage of an extended set of change patterns significantly decreases the time required to perform required changes in EDSOAs compared to the usage of a minimal change patterns set.

The CoCC variable is assessed as the number of non-captured changes in the final model provided by the participants as a solution. The variable is calculated with respect to a desired final model (also called a target model) and a model created by the participants (see below for more details). The time variable is measured by the time that the participants spent on providing a set of operations that should capture desired changes. The participants' experience is measured based on their self-evaluations using a 5 level

Description	Scale type	Unit	Range
UML modelling theory (UML-T)	Ordinal	N/A	5 level Likert scale
UML modelling practice (UML-P)	Ordinal	N/A	5 level Likert scale
Programming theory (Prog-T)	Ordinal	N/A	5 level Likert scale
Programming practice (Prog-P)	Ordinal	N/A	5 level Likert scale
Programming of distributed systems theory (Concur-T)	Ordinal	N/A	5 level Likert scale
Programming of distributed systems practice (Concur-P)	Ordinal	N/A	5 level Likert scale
Group affiliation	Nominal	N/A	Groups G1 (3 patterns), G2 (5 patterns), G3 (4 primitives)

Table 4: Independent Variables

Likert scale. The dependent variables together with their scale types, units, and ranges are shown in Table 3. The independent variables are shown in Table 4.

5.1.2 Study Design

The experiment took place as a part of the Software Architecture lecture at the University of Vienna, Austria, in the Summer Semester 2015. Therefore, it was compulsory for students in the course.

Subjects The subjects of the study were 90 students of the Software Architecture lecture at the University of Vienna.

Objects The objects used in our study are 4 EDSOA models that differ in their complexity, i.e., the number of actors and interconnections among them. Those models are adapted from industrial case studies used in one of our previous projects or constructed to reflect common behaviour and scenarios very similar to those that exist in practice. The following models are used: "Exchange Rates" (Model 1), "ATM Machine" (Model 2), "Travel Booking" (Model 3), and "CRM (Customer Relationship Management) Fulfilment" (Model 4).

Let us consider one of the models used in our study, for example, the model "ATM Machine". Figures 11 and 12 show the source and the target model of the system. The changes between the two models used in the experiment were marked in red so that they can be easily recognized. When a new actor or event appears in the system, its name is marked in red as well as the symbol of an actor whose input or output events are changed. The event flow among the actors is represented by a dashed line with an event name written beside. The actors are represented as square boxes. The actors with arrows inside represent so-called condition actors that send their output events

Model	Description	Actors	Control Flows	Source-Target Changes
Exchange Rates	Converting currencies and finding the best conversion option	Source: 8 Target: 9	Source: 11 Target: 12	19
ATM Machine	Automatic transaction machine to perform financial transactions	Source: 12 Target: 13	Source: 15 Target: 17	22
Travel Booking	Booking a journey including hotel, flight, and car	Source: 12 Target: 13	Source: 17 Target: 20	39
CRM Fulfilment	Managing interactions between a customer and a company by verifying different information	Source: 21 Target: 23	Source: 34 Target: 36	87

Table 5: Studied Models

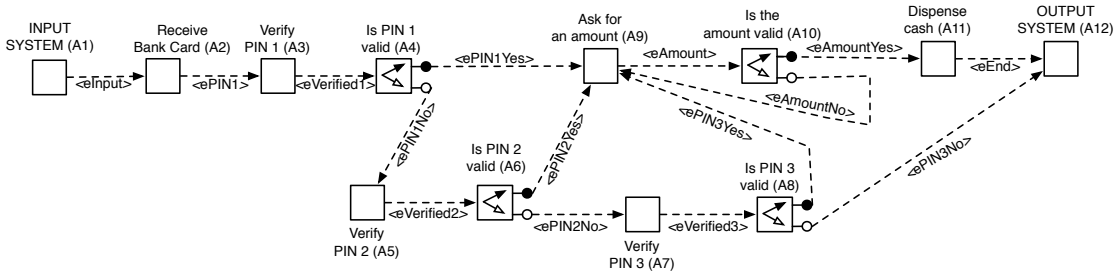

Figure 11: ATM Machine Model – Source Model

depending on the evaluation (TRUE or FALSE). They can send either events related to the TRUE evaluation, e.g. Event "ePIN1Yes" for Actor "A4", or events related to the FALSE evaluation, e.g. Event "ePIN1No" for the same actor (see [40] for more details). The abbreviations of the actors' names (i.e., "A1", "A2", etc., see the figures) are used to shorten the time needed for writing corresponding change operations. The target model of the system is created by performing different types of changes on the source model that can reflect different real situations. For example, insert of a new actor or event, delete some actors or events, replace an actor or event with a new one, add or delete a control dependency, etc. (see [44] for detailed real-world examples). The information related to the Models 1,3, and 4 are summarized in Table 5. The corresponding graphical representations are not shown because of space limitations.

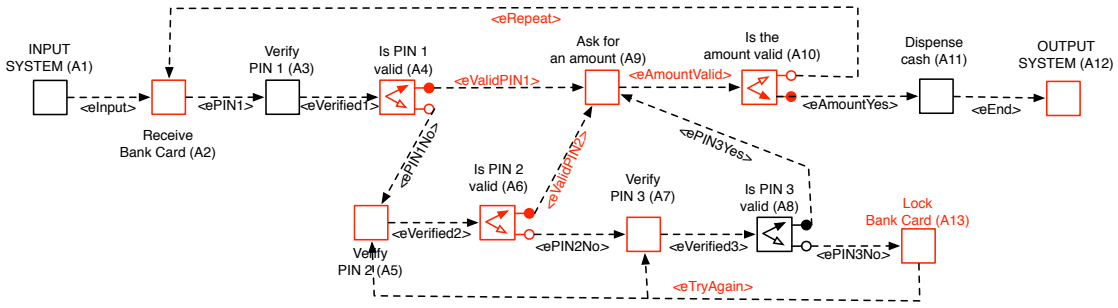

Figure 12: ATM Machine Model – Target Model

Instrumentation All participants were provided with a document containing a detailed description of the notation used in the models. Depending on the group affiliation the document also contained a set of change operations that can be used to transform the given source models into the corresponding target models. In the same document, an introductory example of the source/target model pair is described together with a solution, to get participants more familiar with the task.

In addition, participants were provided with a questionnaire to be filled-in during the study execution. On the first page of the questionnaire, the participants had to rate their experience on a scale of 1–5, i.e., theoretical knowledge as well as practical experience on UML modelling, programming, and programming distributed systems (concurrent programming). Regarding the theoretical knowledge, a scale level 1 means "I have weak theoretical knowledge" and 5 means "I have strong theoretical knowledge". With respect to the practical experience 1 means "I never use it in practice" and 5 means "I use it in practice every day". The subsequent pages contain the 4 source/target model pairs to be studied by participants[3]. The models were shuffled so that 4 different combinations were generated (a fixed sequence of models is hold, only the starting point was variable) and randomly assigned to the participants. The shuffling was used to ensure that we get the more/less balanced data for all the models in terms of equalizing the confounding factors such as possible fatigue effects or the lack of time needed to complete all the models.

We also provided a table where the participants had to enter the time slots during which they studied each of the models. Each time slot contains a start and stop time, indicating the time when the participants started studying the given model and the time when they finish it, respectively. There were more time slots in case the participants wanted to study the model more than one time. The time is written in the format *hour : minute*.

5.1.3 Execution

Preparation The total time for the whole study was 2.5 hours. During the first hour the participants and the experimenters (people who supervised the experiment) studied together the notation and an introductory example, in order to ensure that the participants comprehensively understand the notation and tasks to be studied and to clarify any possible ambiguity and confusion. Two experimenters, who were familiar with the internals of the study, were present during the whole study execution, so that the participants could pose any additional clarification questions.

[3]The number of models to be studied per participant is estimated in a pre-study, conducted with our colleagues, to ensure that the participants have fairly enough time to study all of them.

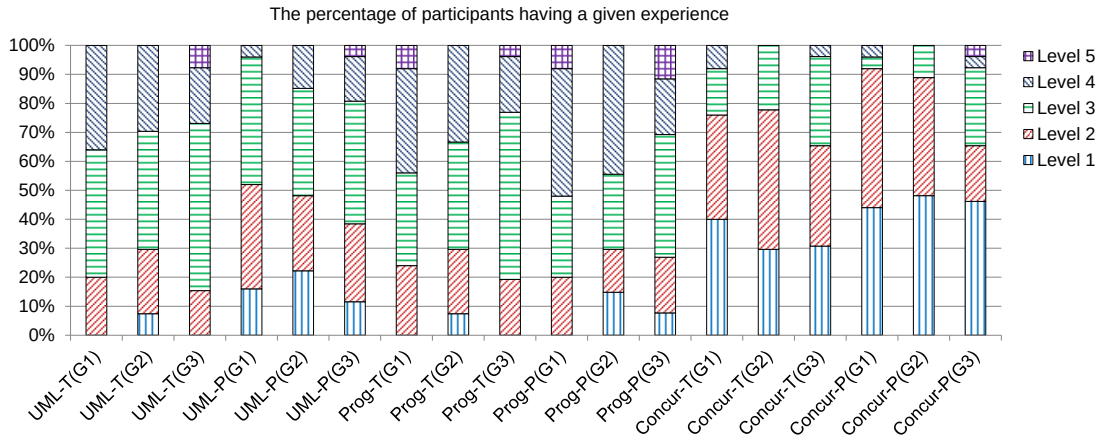

Figure 13: Participants' Demographic Information

Data collection The data related to the participants' demographic information are shown in Figure 13 (please see Table 4 for the abbreviations used in the figure). According to the experience of the participants, we can say that most of the participants have moderate experience (levels 2, 3, and 4). According to their self-evaluations, we can say that they have slightly worse knowledge and experience on programming distributed systems than on UML modelling and programming in general.

Since we did not consider related knowledge and experience as a factor in our study, an influence of the independent variables (primarily participants' experiences) on the dependent variables is eliminated by balancing the characteristics between the given 3 groups of participants. The participants were randomly assigned to the three groups. From Figure 13 we can see that the experience of the participants is quite well balanced among the three studied groups. We also confirmed it statistically by pursuing the Cliff's test and concluded that there is no significant difference in the experience between the 3 groups (see below for details on how to perform the given test).

5.2 Analysis

5.2.1 Descriptive Statistics

Six of the participants (3 from the G1 group and 3 from the G3 group) who wrote nothing or just a few operations in total were excluded from the analysis because this would just introduce bias in the results. The descriptive statistics (mean, median, and standard deviation) related the correctness and time variables for each studied model as well as for all models together, after removals, is shown in Table 6.

As mentioned before, the correctness variable is calculated as the number of non-captured changes in the final models provided by the participants with respect to the desired final models. Particularly, the non-captured changes are calculated per actor and then

Model	Group	Num. of partic.	Mean (CoCC)	Median (CoCC)	St. Dev. (CoCC)	Mean (Time)	Median (Time)	St. Dev. (Time)
Model 1	G1	27	1.74	1	2.79	12.26	11	4.79
	G2	30	3.16	1	4.38	12.60	12	5.25
	G3	27	2.04	2	2.26	12.52	11	5.22
Model 2	G1	27	1.37	1	2.63	17	16	8.53
	G2	30	3.96	1	5.58	15.43	13	9.04
	G3	27	2.14	1	3.42	17.18	15	7.24
Model 3	G1	27	2.14	0	2.99	17.41	15	7.02
	G2	30	5.66	2	8.61	18.86	18	8.61
	G3	27	4.33	1	7.24	19.51	19	7.30
Model 4	G1	27	9.62	4	17.16	24.03	21	8.67
	G2	30	21.46	9	24.83	21.96	23	7.68
	G3	27	14.22	8	15.40	29.85	29	10.36
All models	G1	27	14.88	9	19.14	70.70	72	9.31
	G2	30	34.26	21	35.62	68.86	68	11.35
	G3	27	22.74	15	21.65	79.07	77	11.13

Table 6: Descriptive Statistics

summed up for all actors in the model. The number of non-captured changes per actor is calculated as the number of primitive changes that need to be performed in order to transform an actor in the final model provided by the participants to the corresponding actor in the desired final model. For example, to transform Actor "A10" in Figure 11 to Actor "A10" in Figure 12 4 primitive changes are needed (removing Event "eAmountNo", adding Event "eRepeat", removing Event "eAmount", and adding Event "eAmountValid"). Actor "A13" is new so the number of primitive changes in this case is 3 (adding Actor "A13", adding Event "ePIN3No, and adding Event "eTryAgain"). When all actors in the system are taken into account, the number of changes to be made to transform the source system into the target system is 22. Regarding the other studied models, Model 1, 3, and 4, the number of changes to transform the source into the target system is 19, 39, and 87, respectively.

5.2.2 Testing Hypotheses

To analyse the data obtained in the study, the following statistical tests are performed using the programming language R [28]:

- Normality analysis: The Shapiro Wilk normality test (R function *shapiro.test*)
- Comparison of a location shift between more than two variables: The Cliff's test (R function *cidmulv2*)

The first step in the analysis is examining if our data are normally distributed or not. In case of not normally distributed data, we have to use the non-parametric tests in the next step of our analysis, otherwise we can use the parametric tests [9]. The obtained p-values for the Shapiro Wilk normality test are lower than 0.05 (i.e., the level of confidence is 95 %), which means that our data show significant variation from the normal distribution.

Model 1 (p-val, p-crit, p-hat)			Model 2 (p-val, p-crit, p-hat)			Model 3 (p-val, p-crit, p-hat)			Model 4 (p-val, p-crit, p-hat)			All models (p-val, p-crit, p-hat)		
	G1	G2		G1	G2		G1	G2		G1	G2		G1	G2
G2	0.280 0.025 0.581	--	G2	0.041 0.017 0.640	--	G2	0.290 0.025 0.578	--	G2	0.150 0.025 0.613	--	G2	0.096 0.017 0.634	--
G3	0.150 0.017 0.614	0.690 0.050 0.531	G3	0.640 0.050 0.462	0.058 0.025 0.355	G3	0.190 0.017 0.603	0.900 0.050 0.509	G3	0.130 0.017 0.624	0.650 0.050 0.464	G3	0.150 0.025 0.617	0.590 0.050 0.457

Table 7: The Results of the Cliff's Method - Correctness (p-val<p-crit -> result significance, p-hat -> the effect size)

Model 1 (p-val, p-crit, p-hat)			Model 2 (p-val, p-crit, p-hat)			Model 3 (p-val, p-crit, p-hat)			Model 4 (p-val, p-crit, p-hat)			All models (p-val, p-crit, p-hat)		
	G1	G2		G1	G2		G1	G2		G1	G2		G1	G2
G2	0.780 0.017 0.523	--	G2	0.330 0.025 0.422	--	G2	0.520 0.050 0.551	--	G2	0.490 0.050 0.444	--	G2	0.560 0.050 0.453	--
G3	0.890 0.025 0.511	0.990 0.050 0.501	G3	0.810 0.050 0.519	0.150 0.017 0.613	G3	0.180 0.017 0.613	0.480 0.025 0.557	G3	**0.020** **0.025** **0.687**	**0.004** **0.017** **0.721**	G3	**0.011** **0.025** **0.701**	**0.002** **0.017** **0.744**

Table 8: The Results of the Cliff's Method - Time (p-val<p-crit -> result significance, p-hat -> the effect size)

Therefore, we decided to apply non-parametric tests to test our hypotheses. The Cliff's method in conjunction with the Hochberg's method (to control the probability of one or more type I error) that allows heteroscedasticity (different variances in the tested groups) and performs well when tied values can occur [47] is used.

The results of the Cliff's method for the correctness and time variables are shown in Tables 7 and 8. Particularly, the p-values that show if there is a significant difference between the groups, the corresponding critical p-values, and the p-hat values that measure the effect sizes are shown. If the p-values are lower than the corresponding critical p-values it means that there exists a significant difference between the groups [47]. The effect size indicates how strong is the obtained difference between the groups. Values around 0.556 indicate small effect size, around 0.638 medium effect size, and around 0.714 large effect size [15]. From the obtained results, we can see that there is no significant difference in the correctness among the groups for all studied models separately nor in the case where all models are considered together, when the correctness of all models is summed up. However, for the time variable there exists a significant difference for Model 4 and the case where all models are considered together (see Table 8). The p-values in the bottom of the table indicate a significant difference obtained between Group G1 and Group G3 as well as between Group G2 and Group G3. No significant difference is found between Group G1 and Group G2. The obtained effect sizes for the groups that show significant difference in the time variable can be considered as large, which indicates a strong difference.

5.3 Discussion

In the view of the obtained results, we can conclude that the Hypothesis $\mathbf{H_1}$ of our study is not supported, i.e., the usage of both a minimal and an extended change patterns set does not significantly increase the correctness of the captured changes in EDSOAs compared to the usage of a minimal set of change primitives. Therefore, our results suggest that the abstraction level of the change patterns/primitives does hardly have an influence on the level of correctness achieved. The expectations that change patterns significantly increase the correctness, driven by the correctness-by-construction principle discussed in Section 5.1.1, did not materialize. A possible reason might be that the change patterns are not correctly utilized affecting the appearance of a large number of non-captured changes in the models. By considering the results from all 4 models we find that almost 80% of the participants who used patterns applied them correctly, i.e., they introduced just around 10% of non-captured changes with respect to all required changes.

To better comprehend the results for the Hypothesis $\mathbf{H_1}$, we can actually associate them with the results obtained for the time variable that are related to the Hypothesis $\mathbf{H_2}$ of our study. Namely, in achieving the same or similar level of correctness (as discussed above no significant difference is found), the groups that utilized change patterns (Groups G1 and G2) needed significantly less time to perform all required transformations (for all 4 tasks together) compared to the group that utilized change primitives (Group G3). The same remark applies to Model 4. However, for Models 1, 2, and 3 we found no statistically significant difference in the time variable. For Model 4, the highest number of transformations is required to transform the source model into the target one. Totally 87 primitive changes are required compared to Models 1, 2, and 3, that required 19, 22, and 39 primitive changes, respectively (see Section 5.2.1 for more details). We conclude that a certain level of transformation complexity is needed in order that change pattens provide benefits in performing changes in the system. Consequently, we can say that the Hypothesis $\mathbf{H_2}$ of our study is partially supported, i.e., under the assumption that a certain level of transformation complexity exists, the usage of both a minimal and an extended change patterns set significantly decreases the time required to perform required changes in EDSOAs, compared to the usage of a minimal set of change primitives.

From the obtained results in Tables 7 and 8, we can say that the Hypotheses $\mathbf{H_3}$ and $\mathbf{H_4}$ of our study are not supported, i.e., the usage of an extended set of change patterns neither significantly increases the correctness of the captured changes nor decreases the time required to perform the changes in EDSOAs, compared to the usage of a minimal change patterns set. Therefore, our expectations that two additional, more abstract, change patterns would provide more efficient way of capturing changes did not materialize. To examine the potential usage of the additional patterns, we compare the number

of change operations that need to be written in the case where the extended patterns set is used versus the case where the basic patterns set is used. The additional patterns can provide significant benefits only in the first 3 models where the number of operations to capture the required changes can be reduced from 21 to 16 (23.8% of reduction), using the extended patterns set compared to the basic one. Since the additional patterns can provide benefits only in the first 3 models, we examine the Hypotheses $\mathbf{H_3}$ and $\mathbf{H_4}$ additionally in the case where the first 3 models are considered together. Also in that case, no significant difference in the observed variables is found. An explanation for the obtained results might be that the participants realised that the additional 2 patterns can be captured by the other 3 patterns. Since the additional 2 patterns represent patterns that operate on one actor, that do not provide huge benefits in capturing more changes than other 3 patterns, the participants simply did not use them as an option. We examined this explanation by studying the usage of the additional patterns in the system. Table 9 shows the number of participants (and the corresponding percentage) who used any of the additional patterns. From the obtained results, we can say that a relatively small number of participants used any of the patterns. Furthermore, we examine if the additional 2 patterns were correctly applied in the models, since incorrect pattern usage can increase the number of non-captured changes in the models. The usage of the additional patterns introduced totally 58 non-captured changes in the first 3 models, of which 24 come from the Replace pattern (made by 2 out of 5 participants who used the Replace pattern) and 34 from the Reroute pattern (made by 8 out of 17 participants who used the Reroute pattern). Therefore, we can say that the relatively high number of non-captured changes is introduced by the incorrect usage of the additional patterns set. This fact and the fact that the additional patterns are rarely used, partially support the explanations of the results for Hypotheses $\mathbf{H_3}$ and $\mathbf{H_4}$.

To summarize the obtained results regarding Hypotheses $\mathbf{H_1}$ and $\mathbf{H_2}$, we can say that working on change abstractions at higher level does not provide significant benefits, if the goal is only correctness for smaller amounts of changes, but for performing a lot of frequent changes, higher level abstractions like change patterns make sense. This is quite reasonable, since capturing a small number of changes can be more or less easily tracked using change primitives without affecting the model correctness while change patterns still have to be appropriately combined (especially in the case where the set of changes to be preformed do not comply with a set of change primitives captured in a given pattern). Regarding Hypotheses $\mathbf{H_3}$ and $\mathbf{H_4}$, we can say that the potential of using the additional patterns could not be fully exploited, since the subjects used them only to a limited extent and had troubles with their correct application.

Examining more precisely which patterns in which situations could provide benefits or cause disadvantages in the process of performing changes needs to be further investigated. Also, we plan to examine the obtained results in cases where expert users who are more

	Model 1	Model 2	Model 3	Model 4
Replace pattern	1 (3.33%)	1 (3.33%)	3 (10%)	10 (33.33%)
Reroute pattern	9 (30%)	12 (40%)	7 (23.33%)	8 (26.67%)

Table 9: Number of Participants Who Used Replace or Reroute Patterns

experienced with model transformations as well as more complex patterns that operate on several model elements are used.

Based on our results, we can conclude for research question RQ2 that a higher efficiency in performing changes using change patterns compared to primitives in EDSOAs is indeed supported by our empirical results. However, the usage of a minimal set of change patterns did not significantly increase the efficiency of performing changes in EDSOAs compared to the usage of an extended change patterns, or vice versa. That is, additional research is required to find exactly the right set of change patterns for optimizing our design method for supporting changes in EDSOAs.

5.4 Validity evaluation

In this section we discuss the various threats to validity of our study and how we tried to minimize them:

5.4.1 Conclusion validity

The conclusion validity defines the extent to which the conclusion is statistically valid. The statistical validity might be affected by the size of the sample (27, 30, and 27 students in the groups G1, G2, and G3, respectively). In a between subjects-design, 20 participants are recommended to detect a large effect in the one way ANOVA test with a power of 0.8 and a significance level of 0.05 [4]. In the corresponding non-parametric Cliff's test maximum 15 % more participants can be expected [49] leading to 23. As we obtained that there is a statistically significant difference between the studied groups for the given sample size, we would be able to detect even tiny differences between the groups if the sample size increases. Therefore, there is a low threat to conclusion validity of our results.

5.4.2 Construct validity

The construct validity is the degree to which the independent and the dependent variables are accurately measured by the appropriated instruments. A possible threat to validity might be the measuring of the time variable. The participants could have forgotten to write the time right before they start and right after they finish studying the models which represents a threat to the accuracy of the time variable. In order to reduce that

threat, we wrote a reminder before each studied model to remind the participants to write the stop time in the previously studied model if they forgot it and the start time for the given model they intend to study as the next one.

The interpretation of the answers to the questions might result in a threat to validity of the dependent variable. The change operations written by the participants are thoroughly examined by the first author and additionally checked by the second one. Therefore this potential threat is mitigated to a large degree.

5.4.3 Internal validity

The internal validity is the degree to which conclusions can be drawn about cause-effect of independent variables on the dependent variables. We deal with the following issues:

Differences among subjects. The variation in human performance might distort the results of the study, and then the performance differences would not arise from the difference in treatments. In this particular study, the participants' experience is quite well balanced among the three groups in the study and there is no significant difference among the groups (see Section 5.1.3). Thus, this factor is not seen as a strong threat to validity.

Fatigue effects. Total time limit for the whole study was 2.5 hours so fatigue was not very relevant. Also, the shuffling of the models and a pre-study estimation of the required time for studying the questions (see Section 5.1.2) helped to cancel out these effects.

Measuring method. A potential threat to validity might be that the understanding of the questionnaire could have been biased towards "Group $A_{hierarchy}$". Answering some of the questions might be easier for that group because the architecture for that group reduces the decision space by pointing to the component or the set of components related to the examined concern. However, those questions are based on the established comprehension framework related to examining the relevant concerns of (a part of) the system and how those concerns are interrelated [24]. The established task framework also ensures that many aspects of typical understanding contexts are covered. Beside the usage of the common framework the questions are imaginatively constructed to measure the deeper understanding of the groups (see Section 5.1.2). As a result, the questionnaire concerned both global and detailed knowledge, as well as static and dynamic aspects. Therefore, we consider that this threat is mitigated to a large extent.

5.4.4 External validity

The external validity is the degree to which the results of the study can be generalized to the broader population under study. The following facts are identified: The greater the external validity, the more the results of an empirical study can be generalised to actual software engineering practice.

The studied models. Although we used only 4 models in the study, we consider that a risk for generalizing the results is mitigated to a large extent. In particular, we created different types of changes (without emphasizing particular change types) on the source model that reflect different real situations like inserting of a new actor or event, deleting some actors or events, replacing an actor or an event with a new one, adding or deleting a control dependency, etc. [44]. Therefore, we consider that this threat is mitigated to a large extent. However, examining in how far different change operations are beneficial for models with different characteristics or structures need to be further investigated.

The used notation. Another possible threat to external validity relates to the modelling notation used (i.e., DERA framework). In the given notation, the interfaces of the actors are explicitly defined, which is close to the developers' perception and reduces a non-deterministic behaviour of event-based communications (see Section 3). As mentioned before, the used notation and concepts comply with most existing EDSOAs.

Explicitly defined actors' interfaces, used in the notation, enable that the dependencies between the actors are known at any time. Therefore, the dependencies do not need to be extracted from the source code by studying different event-processing rules (e.g. complex event processing rules [20], publish-subscribe rules [20]). However, in case of no explicit interfaces, where a user has to study actors event processing rules to infer how the actors communicate, different results might be obtained. More studies are necessary to investigate those effects.

Subjects. In our study, we used students who have moderate knowledge and experiences related to the studied problem. It has been shown in previous research that software engineering students may provide an adequate model for the experts population [46]. However, the results cannot be generalized to experts who will presumably be able to apply patterns more effectively.

6 Conclusion

This study presents a change pattern based design method for supporting evolution in EDSOAs, together with a controlled experiment on understanding and performing changes in EDSOAs using change operations at different abstraction levels (i.e., low-level change primitives and high-level change patterns). We have clearly shown in the

feasibility of our approach by developing a design method and tool support using a model-driven tool chain consisting of 3 domain-specific languages. In our empirical study, we investigated the efficiency of transforming a given architecture model into a desired one, using 3 different sets of change operations: a minimal set of 3 change patterns, an extended set of 5 change patterns, and a minimal set of 4 change primitives. Our results indicate that change patterns based evolution is more efficient, i.e., requires a significantly less time to capture a similar level of correctness, compared to the change primitives based evolution. However, the observed improvement in efficiency has only been shown for the model that required the highest number of changes to be captured as well as for the case where all models are considered together. Therefore, we conclude that a certain level of transformation complexity is required for change patterns to provide benefits in performing changes of a system. In addition, no significant difference in efficiency of an extended pattern set compared to a minimal set of patterns is found. With respect to that, we conclude that the additional patterns could not be fully exploited, since the subjects used them only to a limited extent and had troubles with their correct application. The results of our study – besides the empirical evidence – also provide a contribution toward a tool support for performing changes in EDSOAs. In particular, our research informs the choice of change operations that such tools should provide. Future research should include examining more precisely in how far different change operations can support capturing different types of changes in the system as well as how experts would apply different types of change operations.

Acknowledgments

This work was supported by the Austrian Science Fund (FWF), Project: P24345-N23.

References

[1] S. T. andy Uwe Zdun. Domain specific languages for maintaining and analyzing changes in event-based architectures. In *8th International Workshop on Evolutionary Business Processes (EVL-BP 2015), Co-located with the 19th IEEE EDOC Conference*, September 2015.

[2] H. P. Breivold, I. Crnkovic, and M. Larsson. A systematic review of software architecture evolution research. *Information and Software Technology*, 54(1):16–40, 2012.

[3] J. Claes, I. Vanderfeesten, H. Reijers, J. Pinggera, M. Weidlich, S. Zugal, D. Fahland, B. Weber, J. Mendling, and G. Poels. Tying process model quality to the modeling process: The impact of structuring, movement, and speed. In A. Barros, A. Gal, and E. Kindler, editors, *Business Process Management*, volume

7481 of *Lecture Notes in Computer Science*, pages 33–48. Springer Berlin Heidelberg, 2012.

[4] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. L. Erlbaum Associates, 1988.

[5] J. A. Cruz-Lemus, M. Genero, M. E. Manso, S. Morasca, and M. Piattini. Assessing the understandability of uml statechart diagrams with composite states–a family of empirical studies. *Empirical Softw. Engg.*, 14(6):685–719, Dec. 2009.

[6] P. Dadam and M. Reichert. The adept project: A decade of research and development for robust and flexible process support - challenges and achievements. *Computer Science - Research and Development*, 23(2):81–97, 2009.

[7] J. L. Fiadeiro and A. Lopes. An algebraic semantics of event-based architectures. *Mathematical. Structures in Comp. Sci.*, 17(5):1029–1073, Oct. 2007.

[8] L. Fiege, G. Mühl, and F. C. Gärtner. Modular event-based systems. *The Knowledge Engineering Review*, 17(4):359–388, Dec. 2002.

[9] A. Field, J. Miles, and Z. Field. *Discovering Statistics Using R*. SAGE Publications, 2012.

[10] S. Ganesan, Y. Yoon, and H.-A. Jacobsen. NIÑOS take five: the management infrastructure for distributed event-driven workflows. In *5th ACM Int'l Conf. on Distributed event-based system (DEBS)*, pages 195–206. ACM, 2011.

[11] J. Garcia, D. Popescu, G. Safi, W. G. J. 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*, ESEC/FSE 2013, pages 367–377, New York, NY, USA, 2013. ACM.

[12] 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.

[13] M. B. Juric. {WSDL} and {BPEL} extensions for event driven architecture. *Information and Software Technology*, 52(10):1023 – 1043, 2010.

[14] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *Software Engineering, IEEE Transactions on*, 28(8):721–734, Aug. 2002.

[15] H. C. Kraemer and D. J. Kupfer. Size of treatment effects and their importance to clinical research and practice. *Biological Psychiatry*, 59(11):990 – 996, 2006.

[16] 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*, ICSE '10, pages 185–194, 2010.

[17] M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[18] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.*, 1:213–221, Sept. 1984.

[19] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA., Dec. 2001.

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

[21] P. Niblett and S. Graham. Events and service-oriented architecture: The oasis web services notification specifications. *IBM Syst. J.*, 44(4):869–886, Oct. 2005.

[22] M. C. Otero and J. J. Dolado. Evaluation of the comprehension of the dynamic modeling in {UML}. *Information and Software Technology*, 46(1):35 – 53, 2004.

[23] 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.

[24] M. J. Pacione, M. Roper, and M. Wood. A novel software visualisation model to support software comprehension. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 70–79, Washington, DC, USA, 2004. IEEE Computer Society.

[25] J. Pinggera, S. Zugal, M. Weidlich, D. Fahland, B. Weber, J. Mendling, and H. Reijers. Tracing the process of process modeling with modeling phase diagrams. In F. Daniel, K. Barkaoui, and S. Dustdar, editors, *Business Process Management Workshops*, volume 99 of *Lecture Notes in Business Information Processing*, pages 370–382. Springer Berlin Heidelberg, 2012.

[26] 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. ACM, 2012.

[27] H. C. Purchase, L. Colpoys, M. McGill, D. Carrington, and C. Britton. Uml class diagram syntax: An empirical study of comprehension. In *Proceedings of the 2001*

Asia-Pacific Symposium on Information Visualisation - Volume 9, APVis '01, pages 113–120, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.

[28] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2013.

[29] 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.

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

[31] 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, 2008.

[32] N. Russell, A. ter Hofstede, D. Edmond, and W. van der Aalst. Workflow data patterns: Identification, representation and tool support. In L. Delcambre, C. Kop, H. Mayr, J. Mylopoulos, and O. Pastor, editors, *Conceptual Modeling âĂŞ ER 2005*, volume 3716 of *Lecture Notes in Computer Science*, pages 353–368. Springer Berlin Heidelberg, 2005.

[33] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical report, BPMcenter.org, 2006.

[34] 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*, pages 16–30. Springer, 2008.

[35] L. Thom, M. Reichert, and C. Iochpe. Activity patterns in process-aware information systems: Basic concepts and empirical evidence. *International Journal of Business Process Integration and Management (IJBPIM)*, 4(2):93–110, 2009.

[36] D. Tombros and A. Geppert. Building Extensible Workflow Systems Using an Event-Based Infrastructure. In *12th Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*, pages 325–339. Springer-Verlag, 2000.

[37] 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 283–290, Graz, Austria, September 2013. Springer.

[38] 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.

[39] 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.*, pages 44–46, June 2015.

[40] 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.

[41] H. Tran and U. Zdun. Event actors based approach for supporting analysis and verification of event-driven architectures. In *17th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, pages 217–226, USA, September 2013. IEEE Computer Society Press.

[42] W. M. P. van der Aalst, M. Pesic, and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - Research and Development*, 23(2):99–113, 2009.

[43] B. Weber, J. Pinggera, V. Torres, and M. Reichert. Change patterns in use: A critical evaluation. In S. Nurcan, H. Proper, P. Soffer, J. Krogstie, R. Schmidt, T. Halpin, and I. Bider, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 147 of *Lecture Notes in Business Information Processing*, pages 261–276. Springer Berlin Heidelberg, 2013.

[44] B. Weber, M. Reichert, and S. Rinderle-Ma. Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data Knowl. Eng.*, 66(3):438–466, Sept. 2008.

[45] 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.

[46] B. Weber, S. Zeitelhofer, J. Pinggera, V. Torres, and M. Reichert. How advanced change patterns impact the process of process modeling. In I. Bider, K. Gaaloul, J. Krogstie, S. Nurcan, H. Proper, R. Schmidt, and P. Soffer, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 175 of *Lecture Notes in Business Information Processing*, pages 17–32. Springer Berlin Heidelberg, 2014.

[47] R. Wilcox. Chapter 7 - one-way and higher designs for independent groups. In R. Wilcox, editor, *Introduction to Robust Estimation and Hypothesis Testing (Third*

Edition), Statistical Modeling and Decision Science, pages 291 – 377. Academic Press, Boston, third edition edition, 2012.

[48] C. Wohlin. *Experimentation in Software Engineering: An Introduction*. The Kluwer International Series in Software Engineering. Kluwer Academic, 2000.

[49] D. Wolfe. Nonparametrics: Statistical methods based on ranks and its impact on the field of nonparametric statistics. In J. Rojo, editor, *Selected Works of E. L. Lehmann*, Selected Works in Probability and Statistics, pages 1101–1110. Springer US, 2012.

[50] S.-T. Yuan and M.-R. Lu. An value-centric event driven model and architecture: A case study of adaptive complement of {SOA} for distributed care service delivery. *Expert Systems with Applications*, 36(2, Part 2):3671 – 3694, 2009.