Event-Driven Actors for Supporting Flexibility and Scalability in Service-Based Integration Architecture

Huy Tran and Uwe Zdun

Software Architecture Research Group
University of Vienna, Austria
firstname.lastname@unvie.ac.at

Abstract. Service-based software systems are often built by incorporating functionalities from other software systems or platforms. A widely used approach in practice is to introduce an intermediate integration layer for hiding the complexity and heterogeneity of the integrated systems or platforms. However, existing approaches introduce limited support for the flexibility of the integration architecture. It is challenging to alter the integration architecture, e.g., due to some exceptions or unanticipated situations such as peak loads or emergencies, because of rigid dependency structures in the integration architecture defined at design or deployment time. In this paper, we propose DERA as a novel approach that exploits event-driven architecture concepts for enhancing the flexibility and scalability of service-based integration architectures. Our approach provides primitive concepts that can easily be analyzed with tools or be used to depict a current snapshot of the integration architecture using graphical notations close to the intuitive perception of stakeholders. We show the applicability of DERA through an industrial case study in the field of software platform integration and evaluate the scalability of our approach.

Keywords: Event-driven architecture, event actors, services, service-based integration architecture, flexibility, scalability, substitutability.

1 Introduction

Service-oriented architectures (SOA) provide efficient means for exposing functionality of software systems or platforms in terms of services with standardized interfaces. Service-based systems can be built by incorporating services provided by other systems or platforms. Instead of directly dealing with the heterogeneity and variety of the integrated systems or platforms, an intermediate integration layer is often introduced for bringing systems or platforms into the service-oriented world and providing the required service integration and composition logic [15]. Figure 1 illustrates a high-level overview of a simplistic platform integration layer design. There is a considerable amount of existing approaches for realizing the integration layer, such as using composite services, enterprise service buses, messaging infrastructures, or business processes.

A certain flexibility is often required at the level of the integration layer to not only support rapid tailoring and customization of the integration architecture but also enable the ability to deal with some exceptions or unanticipated situations such as peak loads

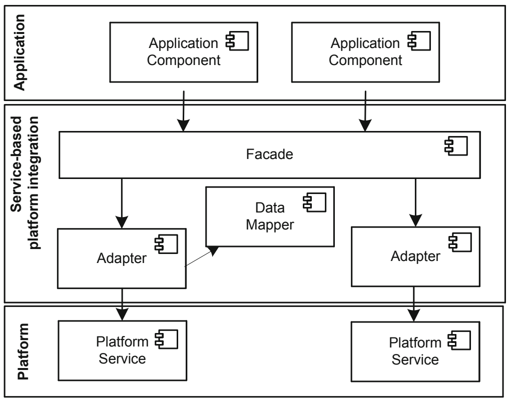


Fig. 1. Overview example for service-based platform integration architectures

or emergencies. The mentioned existing approaches introduce only limited support for the flexibility of the integration architecture through dynamic bindings of the constituting elements (e.g., dependency injection or aspect weaving techniques are used in some of the mentioned approaches) [7,9,8]. Altering the integration architecture is very challenging because of the rigid dependency structures in the integration architecture defined at design or deployment time. Such rigid dependencies also might cause scalability problems as sophisticated algorithms and coordination techniques are required to efficiently schedule and distribute integration and composition logic [14,4].

Event-driven communication styles are potential solutions for facilitating high flexibility, scalability, and concurrency of distributed systems [18]. Due to the intrinsic loose coupling of the participants, the event-based architectural style is used in many large-scale distributed software systems today. The advantages of event-driven communication styles have been extensively exploited in a considerable amount of work proposing different combinations of event-driven architectures (EDA) with business process management and SOAs [12,25]. To the best of our knowledge, none of the existing approaches has exploited EDA for resolving the aforementioned problems in service-based integration architectures. Another issue hindering the use of EDA is that software architects and developers often find the event-driven communication style unintuitive (compared to other paradigms such as remote procedure calls or messaging) and large architectures with many event actors and numerous events hard to comprehend.

In this paper, we propose DERA as a novel approach leveraging the event-driven architecture style to enable flexibility of integration architectures for supporting various kinds of runtime evolution and dynamic adaptation while minimizing the non-deterministic nature of event-based applications. In particular, our approach encapsulates constituting elements (e.g., components, functions, adapters, proxies) using event actors with explicit interfaces. It exploits the event-based communication style to loosen the dependencies among the actors. The interfaces of the actors, described in terms of incoming and outgoing events, are formally specified and constrained to, on the one hand, enable the support for changing actors at runtime (e.g., replacing actors or changing their execution order). On the other hand, well-defined interfaces of actors also reduce the non-deterministic behavior in EDAs. The proposed graphical notations of the DERA concepts can be used to visually depict a current snapshot of the event-driven service integration architecture that is closer to the stakeholders' perception than studying only the event actors' inputs and outputs and their event processing rules. In this paper, we present concepts and formalizations to establish the foundation of our approach as well as a prototypical implementation. We show the applicability of our approach using an industrial case study in the field of service platform integration. Finally, we evaluate the performance and scalability of our approach, as good scalability is one of the central promises of EDAs and a central integration layer is a potential bottleneck in an integration architecture. We can show that our approach offers linear scalability and has only a moderate performance overhead compared to a hard-coded Java implementation with rigid dependencies among the actors.

The paper is structured as follows. Section 2 introduces the case study. We present the DERA (Dynamic Event-driven Actors) approach in Section 3. In Section 4, we describe a prototypical implementation of DERA, revisit the case study and elaborate on how DERA can help to improve the flexibility in the case study, and report on the evaluation of DERA's performance and scalability. We compare to related work in Section 5, and finally we conclude and discuss future work in Section 6.

2 Case Study

We illustrate the application of the concepts presented in this paper in an integration scenario of a warehouse operator application extracted from an industrial case study in the field of service platform integration[1]. In the context of this application, there are three different domain-specific service platforms, namely, a yard management system (YMS), a warehouse management system (WMS), and a remote monitoring system (RMS) that provide a wide variety of services. The warehouse operator application shall utilize these services to perform various necessary tasks that are triggered by events such as the arrival of a truck carrying products to be stored in the warehouse. The three platforms shall be integrated using an service-based platform integration layer akin to the one in Figure 1, and the warehouse operator application shall use the services of the integration layer to access the services of all three platforms.

We present in Figure 2 a schematic view of one scenario of the warehouse operator application. The lifelines of the sequence diagram represent the proxy components which are responsible for interacting with the service platforms involved in the scenarios. One of the crucial requirements for the scenario is that we want to be able to alter the composition logic of the warehouse operator in a flexible manner at runtime. For instance, the warehouse operator might not need to perform `getFreeDock` because the arriving truck is specially assigned to a dedicated dock in the yard. Another example is that the warehouse operators need to call the warehouse staff to prepare for unloading products in the truck. We will revisit and analyze this scenario in Section 4.

[1] `http://indenica.eu/fileadmin/INDENICA/user_upload/d51-casestud.pdf`

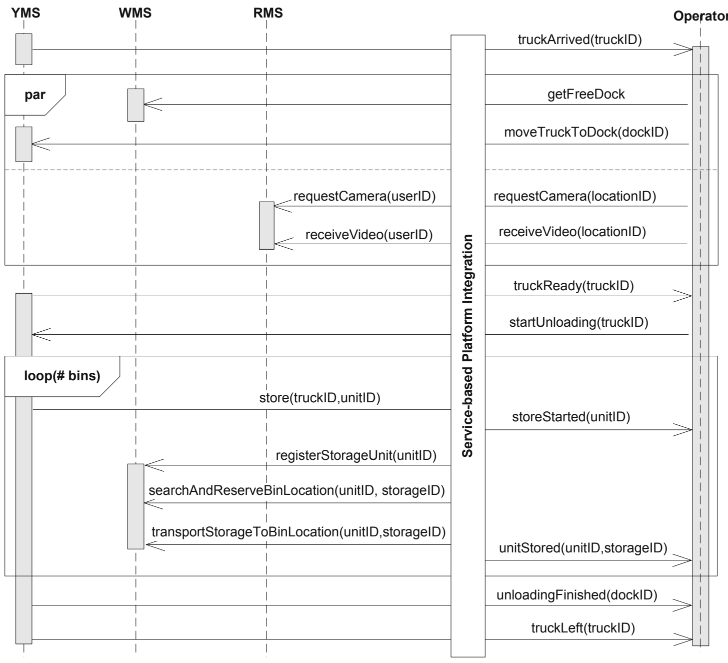


Fig. 2. The behavior of the unloading scenario

3 Dynamic Event-Driven Actors (DERA)

3.1 DERA Primitive Concepts

Figure 3 depicts the meta-model of the primitive concepts forming a DERA system. The semantics of these basic event actors are provided in Table 1. The central notions of DERA are *events* and *event actors*. An event can be considered essentially as “any happening of interest that can be observed from within a computer” [20] (or a software system). An example of an event from the business domain is the arrival of a purchase order. An event is associated with one or more `ObjectReferences` for obtaining references to data sources managed in `ObjectPools`. Events may also have some attributes such as their unique identifier, correlation identifiers, timing attributes, and so on.

An *event type* is a representation of a class of events that share a common set of attributes. An *instance* of an event type is a concrete occurrence of that event type that has a unique identifier and is instantiated with concrete values of the event type’s attributes. An example of an event type are incoming customer orders, whilst the corresponding instance of this type is an incoming order from Alice. Given a concrete event e, `typeof`(e) will be used to denote the event type of e.

The encapsulation of a particular computational unit (e.g., an executable function, a component, a proxy, or an adapter) that performs a concrete task, for instance, executing composition logic of a number of service invocations, performing the role of a service

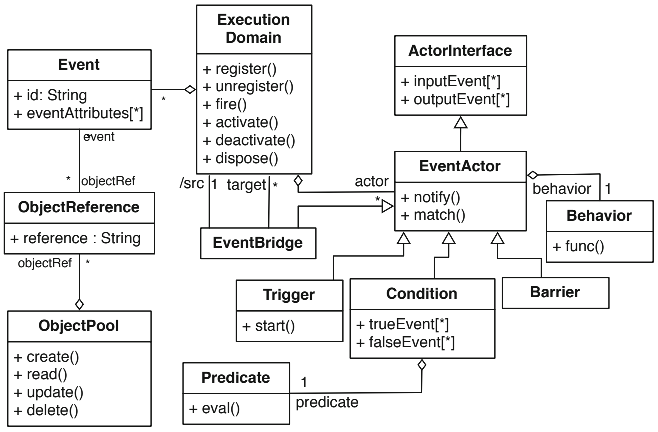


Fig. 3. Meta-model for the primitive DERA concepts and their relationships

adapter or proxy, accessing and transforming data, to name but a few, is an *event actor* (or *actor* for short). Each actor has a well-defined interface represented by the `ActorInterface` class. The actor's interface defines a set of events that the actor awaits (aka input events) and a set of events that the actor will emit after finishing its execution (aka output events). Particular behavior of each actor is defined through a concrete instance of the `Behavior` class. The execution of an actor is triggered by the arrival of any of the input events. At the end of its execution, the actor will emit *all* of its output events that, in turn, may trigger the executions of other actors.

In Table 1, we present the graphical notations of the DERA actors that can be used to visually depict a snapshot of a DERA system at a particular point in time. A formal behavior definition for each type of actor is given as a Finite State Machine (FSM). The transitions between two states represents the occurrence of DERA operation invocations such as `(un)register`, `(de)activate`, `notify`, `match`, `func`, `fire`, and so forth, (see Definition 2 below). Any snapshot of a DERA system can easily be transformed into an FSM or another formal representation (like a Petri Net) to perform formal analysis of the system snapshot. The DERA prototype is based on Java. Developing DERA applications using the Java APIs is rather tedious as they offer a lower abstraction level than DERA system models. For this reason, we additionally provide DeraDSL, a domain-specific language (DSL) for supporting the model-driven specification of DERA systems. The second column of Table 1 shows the code required for definingeach of the actor type in DeraDSL. Note that DeraDSL's constructs will be mapped to the meta-model depicted in Figure 3 and consequently to the Java implementation of DERA for execution. We explain DeraDSL in detail in Section 4.1.

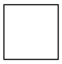
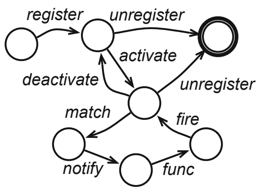
Event actors are defined based on well-defined event interfaces:

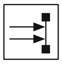
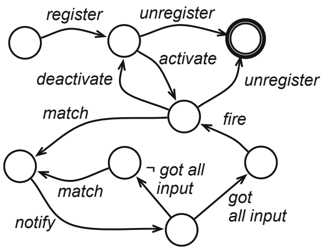
Definition 1 (Event actor interface). *An interface $\mathcal{I}_x$ of a DERA event actor x can be described by a 2-tuple $(\bullet x, x\bullet)$, where $\bullet x$ is a set of input events expected by x and $x\bullet$ is a set of output events to be emitted by x ($\bullet x$ and $x\bullet$ can be empty sets).*

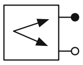
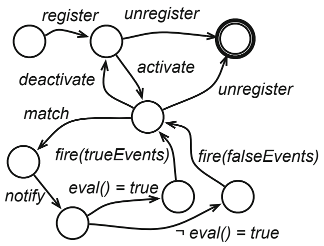
The benefits of specifying well-defined interfaces for DERA event actors are manifold. Firstly, they enable us to *conceptually* capture a snapshot of current state of the DERA

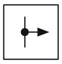
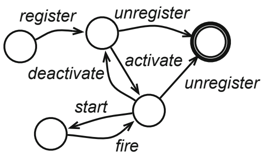
Table 1. Notations and formal definitions of behaviors of DERA actors

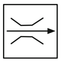
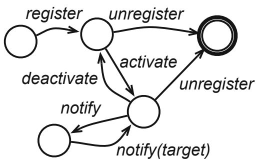
Actor	Notation	DeraDSL construct	Formal behavior definition

`EventActor`		`EventActor <name>`
		`  input [inputEvents]`
		`  output [outputEvents]`
		`  func [actor-behavior]`
			 register
			unregister
			activate
			deactivate
			unregister
			match
			fire
			notify
			func

`Barrier`		`Barrier <name>`
		`  input [inputEvents]`
		`  output[outputEvents]`
			 register
			unregister
			activate
			deactivate
			unregister
			match
			fire
			¬ got all input
			match
			got all input
			notify

`Condition`		`Condition <name>`
		`  input[inputEvents]`
		`  when-true[trueEvents]`
		`  when-false[falseEvents]`
		`  eval [predicate]`
			 register
			unregister
			activate
			deactivate
			unregister
			match
			fire(trueEvents)
			fire(falseEvents)
			eval() = true
			notify
			¬ eval() = true

`Trigger`		`Trigger <name>`
		`  output [outputEvents]`
			 register
			unregister
			activate
			deactivate
			unregister
			start
			fire

`EventBridge`		`EventBridge <name>`
		`  target [targetDomains]`
			 register
			unregister
			activate
			deactivate
			unregister
			notify
			notify(target)

system and derive a directed graph that comprises event actors connected via their inputs and output events at design time or runtime. As a result, we are able to support monitoring and analysis of important properties such as reachability (safety or deadlock checking), liveliness, performance, quality of services, of DERA systems described in terms of such graphs. On the other hand, well-defined interfaces also enable us to support changes at runtime, e.g., substituting an event actor by another with a compatible interface or changing the execution order of event actors by substituting an event actor by another having the same input events but different output events.

3.2 DERA Architecture

Figure 4 shows an overview of the DERA tool-chain on the left-hand side and the DERA runtime architecture on the right-hand side. DeraDSL will be described in Section 4.1. In this section, we focus on the DERA runtime architecture.

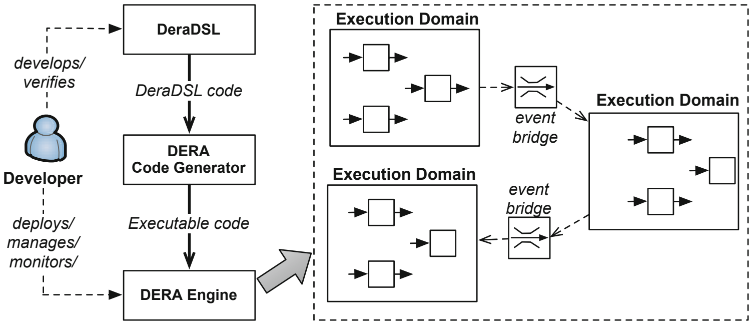


Fig. 4. Overview of DERA development toolchain and system architecture

DERA is designed so that each event actor only concentrates on its own task, its well-defined interfaces defining its input events and the events it is going to emit. There are no tight dependencies between two particular event actors except "*virtual connections*" established via the event-based communications. This is realized using the notion of *event channels* which are abstractions used for delivering events among communicating parties. All actors in an execution domain of a DERA system are connected to the same channel, and all events are published via the channel. All events published on a channel are consumed by all actors registered for the channel. Hence, actors are loosely coupled. This loose coupling leads to the flexibility and scalability of DERA. That is, event actors can be distributed for better load balancing or performance optimization purposes without requiring any sophisticated distribution algorithms.

Event channels are also used as a means to implement logical *execution domains*. Execution domains are useful for supporting runtime governance activities such as deployment, management, and monitoring, as they can be used to group event actors and events. An execution domain might host one or many DERA applications while a certain DERA application can span across several execution domains. Two DERA execution domains can be connected by `EventBridges` which are special event actors responsible for forwarding events from a DERA execution domain to another (see Figure 4). The original function of event bridges may also be extended with extra features such as enriching or transforming the content of events.

Definition 2 (DERA system). *A DERA system $\mathcal{S}$ can be described by a 4-tuple $(\mathcal{E}, \mathcal{A}, \mathcal{C}, \mathcal{O})$, where*

1. *$\mathcal{E}$ is a finite set of events.*
2. *$\mathcal{A}$ is a finite set of event actors. Each event actor $x \in \mathcal{A}$ has a well-defined interface $\mathcal{I}_x(\bullet x, x\bullet)$, where $\bullet x \subseteq \mathcal{E}$ and $x\bullet \subseteq \mathcal{E}$ and can perform a particular function.*
3. *$\mathcal{C}$ is an event channel that is responsible for delivering an event received from a certain event actor exactly once to all other actors registered to the channel. An event actor $x \in \mathcal{A}$ consumes an incoming event $e \in \mathcal{E}$ from the channel $\mathcal{C}$ only if $\mathbf{match}(x, e)$=`true` (for definition of $\mathbf{match}()$ see below). The channel is assumed to be reliable, i.e., no message is lost or altered.*
4. *$\mathcal{O}$ is a set of basic operations, including (but not limited to)*
   - $\mathbf{register}(x)$*, where $x \in \mathcal{A}$, indicates that the actor x is registered to the event channel $\mathcal{C}$.*
   - $\mathbf{unregister}(x)$*, where $x \in \mathcal{A}$, indicates that the actor x unregisters to the event channel $\mathcal{C}$.*
   - $\mathbf{fire}(x, E)$ *or* $\mathbf{fire}(x, e)$*, where $x \in \mathcal{A}$, $E \subseteq \mathcal{E}$, and $e \in \mathcal{E}$, indicates that the actor x fires a set of events E or a single event e, respectively.*
   - $\mathbf{match}(x, e)$*, where $x \in \mathcal{A}, e \in \bullet x$, returns* `true` *if the event e matches the interface of the actor x and* `false` *otherwise.*
   - $\mathbf{func}(x)$*, where $x \in \mathcal{A}$, denotes a concrete task or an arbitrary user-defined behavior of the event actor x.*
   - $\mathbf{deactivate}(x)$ *and* $\mathbf{activate}(x)$*, where $x \in \mathcal{A}$:* $\mathbf{deactivate}(x)$ *is used for putting the execution of x on hold and* $\mathbf{activate}(x)$ *is used for resuming its execution, for instance, after a* $\mathbf{deactivate}(x)$.

The interface operations `register()` and `unregister()` are mainly used for the management of a DERA execution domain, and `deactivate()` and `activate()` deal with lifecycle management. The operation `func()` is the placeholder where one can put in a certain user-defined behavior such as invoking a service, accessing a database, or opening and reading a local file. Please note that the execution of the operation `func()` is not allowed to change the event actor's interface (i.e., input and output events) or emit new events. The main goal of this constraint is to reduce the non-deterministic nature of the event-based communication styles employed in DERA. Changing an event actor's input and output events must be explicitly declared through its interface. As a result, we can *conceptually* establish the dependencies between event actors by observing their interface descriptions. The dependencies can be used for many important tasks such as monitoring and verifying properties of DERA systems and applications.

Definition 3 (Event matching). *Given a DERA system $\mathcal{S}$ described by the 4-tuple $(\mathcal{E}, \mathcal{A}, \mathcal{C}, \mathcal{O})$. Let x be an event actor ($x \in \mathcal{A}$) having an interface $\mathcal{I}_x = (\bullet x, x\bullet)$. An event $e_1 \in \mathcal{E}$* matches *the interface $\mathcal{I}_x$ if and only if there exists at least one event $e_2 \in \bullet x$ such that e_1 and e_2 are of the same event type (i.e., `typeof(`e_1`)` = `typeof(`e_2`)`).*

Definition 4 (DERA application). *A DERA application Φ running in a DERA system $\mathcal{S}$ can be described by a 4-tuple $(A, E, E_{start}, E_{finish})$, where*

- *$A \subseteq \mathcal{A}$ is a finite set of event actors constituting the functionality of the application;*
- *$E \subseteq \mathcal{E}$ is a finite set of events published and consumed by the event actors of $\mathcal{A}$;*
- *$E_{start} \subseteq \mathcal{E}$ is a finite set of events that indicate the start of the application;*
- *$E_{finish} \subseteq \mathcal{E}$ is a finite set of events that indicate the end of the application.*

```
module eu.indenica.casestudy.yms
domain YMS {
   Trigger y1 output [ymsTruckArrived]
   EventActor y2 input [facadeMoveTruckToDock] output [ymsMoveTruckToDockFinished]
      func [MoveTruck]
   EventActor y3 input [ymsMoveTruckToDockFinished] output [ymsTruckReady]
      func [CheckTruckReadyForUnloading]
   Barrier y4 input [ymsTruckReady, facadeStartUnloading] output [ymsStartedUnloading]
   EventActor y5 input [ymsStartedUnloading, ymsUnloadingNotFinished] output [ymsStore]
      func [StoreUnit]
   Condition y6 input [ymsStore] when-true [ymsUnloadingFinished]
      when-false [ymsUnloadingNotFinished]
   EventActor y7 input [ymsUnloadingFinished] output [ymsTruckLeft] func [CheckTruckInDock]
   Application YMS start-with [ymsTruckArrived] end-with [ymsTruckLeft]
}
```

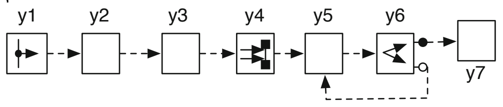


Fig. 5. DERA application and its corresponding graphical representation

We present in Figure 5 an excerpt of the composition logic of the proxy component of the integration architecture that is responsible for interacting with the YMS platform in the scenario shown in Figure 2. The proxy component is described using the programming constructs provided by DeraDSL. Given the specification of the actors and their interfaces, we can build an intuitive graphical representation of a snapshot of the application using the notation from Table 1. Note that the dashed lines among event actors are not real dependencies but virtual connections achieved by analyzing the inputs and outputs of the actors captured in the snapshot of the application.

3.3 Event Actor Substitution

There are several studies in programming languages (especially object-oriented programming) and component-based systems, on the substitutability of data types, objects, and components [22,17]. The event actor substitution in approach is based on the well-known Liskov substitution principle [17]. There are three crucial features of DERA actors that enable us to substitute a certain actor at runtime. Firstly, each actor x explicitly exposes a well-defined event-based interface $\mathcal{I}_x(\bullet x, x\bullet)$. Secondly, the interactions among actors are loosely coupled through event-based communication. Thirdly, the encapsulation of a computational unit that performs a concrete task (i.e., the operation `func()` of an `EventActor`) is not allowed to alter the interface. These features allow us to substitute an actor by one or a set of other actors that introduce (1) an interface which is *compatible* to the original actor's interface (called strong substitution below) or (2) a different interface (called weak substitution below).

Strong substitution occurs when the developers have to replace or upgrade an existing functionality with a different (e.g., better or bug-fixed) version while preserving the overall structure and behavior of the DERA application. It is achieved by defining a new actor y with the same interface as x. x can be replaced with y using the operation `deactivate(x)` to temporarily put the execution of x on hold, and using the operations `register(y)` and `activate(y)` to enable the execution of y.

Definition 5 (Strong substitution). *Let x be an event actor having an interface $\mathcal{I}_x(\bullet x, x\bullet)$. An event actor y posing an interface $\mathcal{I}_y(\bullet y, y\bullet)$ is said to be a* strong substitution *for x if all the following constraints are satisfied:*

1. *Type requirement: y must be of the same type or a subtype of x. That is, if x is of type `EventActor`, then y must be of type `EventActor` or a subtype of `EventActor`.*
2. *Input requirement (applied for every event actor x, such that $\bullet x \neq \varnothing$): $\bullet y = \bullet x$, i.e., y has to be able to accept the same input events as x does.*
3. *Output requirement (applied for every event actor x, such that $x\bullet \neq \varnothing$): $y\bullet = x\bullet$, i.e., y has to fire the same output events as x does.*

Weak substitution occurs when the developers want to alter the structure and behavior of one or all running instances of a DERA application, for instance, skipping some tasks to deal with exceptions or unanticipated circumstances such as peak loads and emergencies or adding new functionalities. This is difficult to achieve with many existing integration architectures due to rigid dependency structures. We can support the required flexibility in DERA by substituting existing event actors with newly defined event actors posing different interfaces. This kind of substitution is called *weak substitution* as it is not going to preserve the original structure and behavior.

Definition 6 (Weak substitution). *In a DERA application $\Phi = (A, E, E_{start}, E_{finish})$, let $x \in A$ be an event actor having an interface $\mathcal{I}_x(\bullet x, x\bullet)$. A* weak substitution *$y$ for x can be achieved by relaxing one or more of the conditions for* strong substitutions.

To illustrate possible relaxations of strong substitution conditions, let us consider the following examples:

1. Type requirement: x and y can be instances of a) the same or b) different types.
2. Input requirement (applied for every event actor x, such that $\bullet x \neq \varnothing$): there are no constraints on the input, but a potential case may be one of the following:
   a) $\bullet x \subseteq \bullet y$, i.e., y is able to be triggered by more input events than x,
   b) $\bullet y = \bullet x \cap \left(\bigcup z\bullet, \forall z \in A \land z \neq x\right)$, i.e., y only considers to accept a subset of x's input events that are going to be emitted by other event actors,
   c) $\bullet x \cap \bullet y = \varnothing$
3. Output requirement (applied for every event actor x, such that $x\bullet \neq \varnothing$): there are no constraints on the output, but a potential case may be one of the following:
   a) $x\bullet \subseteq y\bullet$, i.e., y can emit more events than x.
   b) $y\bullet = x\bullet \cap \left(\bigcup \bullet z, \forall z \in A \land z \neq x\right)$, i.e., y only considers to fire a subset of x's output events that are going to be consumed by other event actors.
   c) $x\bullet \cap y\bullet = \varnothing$

Weak substitutions lead to different levels of changes ranging from light or moderate adjustments (e.g., 2(a), 2(b), 3(a), and 3(b)) to significant and disruptive alterations (e.g., 2(c) and 3(c)) of event actors' interfaces. In some situations, these changes may become undesirable as they can result in anomalies such as dead tasks, deadlocks, or livelocks. It would be unrealistic to require the developers to ensure that a certain substitution must lead to an expected and sound state of the running DERA applications. Instead, exploiting powerful reasoning mechanisms based on formal methods such as process algebras [16,19] or Petri-nets [21] can help developers to analyze a certain runtime snapshot of DERA applications to detect potential flaws and correct the substitution before applying it. Also, existing approaches on behavior inheritance [5] or on using change patterns for preserving certain system properties [24] can be leveraged in the context of DERA to enhance the soundness of actor substitutions. Studying the applicability of those mechanisms is one of our planned future works.

4 Implementation – Case Study Revisited – Evaluation

4.1 DERA Implementation

A prototypical implementation of the DERA concepts has been developed to show the feasibility of our approach and implement the case study presented in the next section. In our implementation, we have defined an abstraction layer covering all primitive concepts of DERA presented in Figure 3. This layer is independent from the underlying technologies. The implementation layer realizes the concepts of the abstraction layer using a particular technology, in our case the Java Concurrency Utilities packages[2]. These packages, which are available in Java JDK 1.5 and later, provide sufficient concurrency utilities. In particular, in our DERA implementation, the event-driven communication style is realized using asynchronous event callbacks. The creation and execution of event actors are managed using the built-in `ExecutorService` with fixed thread pools and the synchronization among actors is done through the `synchronized()` construct.

The DERA `EventChannels` represent a means for delivering events among communication parties and can be realized by asynchronous communication styles. Thus, it is possible to incorporate existing powerful libraries and frameworks such as Java Message Service[3] or PADRES [11] in the DERA implementation. For the evaluation purpose presented in the next section, we opted for implementing DERA `EventChannels` based on pure Java asynchronous multi-threading. Each `EventChannel` has a number of event distributors mapped to a pool of light-weight Java threads. These event distributors receive and deliver events following a round-robin scheduling strategy.

The DERA `EventBridges` used to enable DERA systems to support distributed event processing (see Figure 4) have been realized using REST Web services. REST services are a good match for DERA's flexible and scalable architecture because of the use of the scalable and stateless concepts of the Web in the REST architectural style. In the DERA implementation, the REST services are used for transmitting events between two (possible distributed) execution domain A and B that are connected via an

[2] `http://docs.oracle.com/javase/1.5.0/docs/guide/concurrency/index.html`

[3] `http://jcp.org/en/jsr/detail?id=343`

EventBridge. The EventBridge is realized using a REST Service that is connected as an event actor to domain B. An event actor on domain A acts as a REST client forwarding all events raised within A to the REST service. The REST service delivers all received events to B. A bidirectional bridge can be established by adding a second REST service to the event bridge actor on channel A.

As mentioned before, DeraDSL has been developed aiming at minimizing the noise of Java language structures and supporting efficient DERA application development. We leverage Xtext[4], which is a powerful framework supporting textual language development with several advanced features, for instance, syntax coloring, code completion, validation, excellent integration with Java, and many others, to implement DeraDSL. Furthermore, combining Xtext with Xtend[5] helps us on mapping DeraDSL's elements onto the Java-based constructs and libraries that implement DERA. We partially described some primary elements of DeraDSL in Table 1 and used them to illustrate the development of DERA applications in Figure 5.

4.2 Case Study Revisited

Developing the warehouse operator application, motivated in Section 2, using the DERA prototype is fairly straightforward. First, we need to define an EventActor for encapsulating each task to be carried out. Then, we need to assign the input and output events of that actor. A Barrier may be necessary when we need to wait for more than one event arriving before some other tasks can be performed. For decisions, such as checking if there are enough storage locations in the warehouse to store all units, Condition is used. We show an excerpt of the DeraDSL code defining the warehouse operator composition logic using DERA actors in Listing 1.1. In this excerpt, the concrete definitions of the operation func() of the event actors are in a separate modules and can be cross-referenced. Thus, these functions are omitted in the code. The management operations such as (de)register and (de)activate are also not visible but we will discuss them in the next section. The warehouse operator requires some events from the integration facade, namely, PlatformFacade, defined in the lower part of the code excerpt. Thus, we create an event bridge named OperatorToFacade for delivering events from the WarehouseOperator domain to the PlatformFacade domain. Likewise, the event bridge FacadeToOperator is defined in the PlatformFacade domain for sending events back.

Even though DeraDSL can help on better formulating DERA elements, the code shown in Listing 1.1 is still not close to the developers' perception. At a certain stage, for instance, after finishing development or before deploying, a snapshot of a DERA system and application can be taken and visually depicted using the graphical notations described in Table 1. Accordingly, we can establish an equivalent intuitive graphical representation, as shown in Figure 6, of the aforementioned code. The events from the PlatformFacade domain to the WarehouseOperator domain are shown for illustrating the relationship between two domains. We can see that the semantics of DERA concepts and notations are close to that of traditional conditional structures in existing

[4] http://xtext.org
[5] http://xtend-lang.org

programming languages or widely-used formal models such as such as process algebras [16,19] or Petri-nets [21]. As a result, existing formal analysis techniques can be leveraged (see Section for details).

```
module eu.indenica.casestudy.warehouse.operator
domain WarehouseOperator {
    EventActor TruckArrivedNotified input [facadeTruckArrived] output [
      operatorTruckArrivedNotified]
    EventActor GetFreeDock input [operatorTruckArrivedNotified] output [operatorGetFreeDock]
    Barrier b1 input [facadeGetFreeDockFinished, operatorGetFreeDock] output [
      operatorGetFreeDockFinished]
    EventActor MoveTruckToDock input [facadeGetFreeDockFinished] output [
      operatorMoveTruckToDock]
    EventActor RequestCamera input [operatorTruckArrivedNotified] output [
      operatorRequestCamera]
    Barrier b2 input [operatorRequestCamera, facadeRequestCameraFinished] output [
      operatorRequestCameraFinished]
    EventActor VideoReceiving input [operatorRequestCameraFinished] output [
      operatorReceiveVideo]
    Barrier b3 input [operatorMoveTruckToDock, facadeMoveTruckToDockFinished,
      operatorReceiveVideo, facadeReceiveVideoFinished,facadeTruckReady] output [
      operatorTruckReadyNotified]
    EventActor StartUnloading input [operatorTruckReadyNotified] output [
      operatorStartUnloading]
    Barrier b4 input [operatorStartUnloading, facadeStoreStarted] output [
      operatorStoreStartedNotified]
    EventActor StoringMonitoring input [operatorStoreStartedNotified,
      operatorStoringNotFinished] output [operatorStoringMonitoring]
    Barrier b5 input [operatorStoringMonitoring, facadeUnitStored] output [operatorUnitStored
      ]
    Condition isStoringFinished input [operatorUnitStored] when-true [operatorStoringFinished
      ] when-false [operatorStoringNotFinished]
    Barrier b6 input [operatorStoringFinished, facadeUnloadingFinished, facadeTruckLeft]
      output [operatorFinished]
    EventBridge OperatorToFacade target [eu.indenica.casestudy.facade.PlatformFacade]
    Application WarehouseOperator start-with [facadeTruckArrived] end-with [operatorFinished]
}
module eu.indenica.casestudy.facade
domain PlatformFacade {
    ...
    EventBridge FacadeToOperator target [eu.indenica.casestudy.warehouse.operator.
      WarehouseOperator]
    ...
}
```

Listing 1.1. Excerpt of the code defining the warehouse operator application

4.3 Event Actor Substitutions

In this section, we illustrate weak substitutions for the change requirements introduced in Section 2. The first requirement is to to skip the execution of the task `GetFreeDock`. This can be achieved through a weak substitution, i.e., by defining a new actor `MoveTruckToDockNew` that directly consumes the event `operatorTruckArrivedNotified` emitted by the actor app, as illustrated in Listing 1.2.

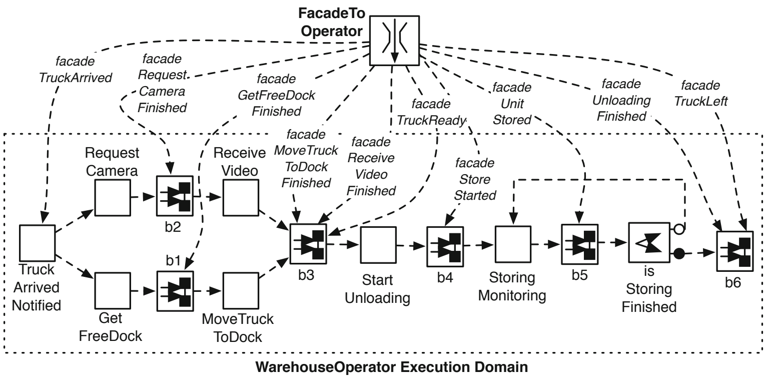


Fig. 6. The graphical representation of the warehouse operator application

```
EventActor MoveTruckToDockNew input[operatorTruckArrivedNotified] output[
operatorMoveTruckToDock]
register [MoveTruckToDockNew] // register the new actor
deactivate [MoveTruckToDock] // temporarily suspend the old actor
deactivate [b1] // we do not need this Barrier
... // verifications can be performed here to detect potential anomalies
activate [MoveTruckToDockNew] // now we can activate the new actor
```

Listing 1.2. Skipping the existing event actor `GetFreeDock`

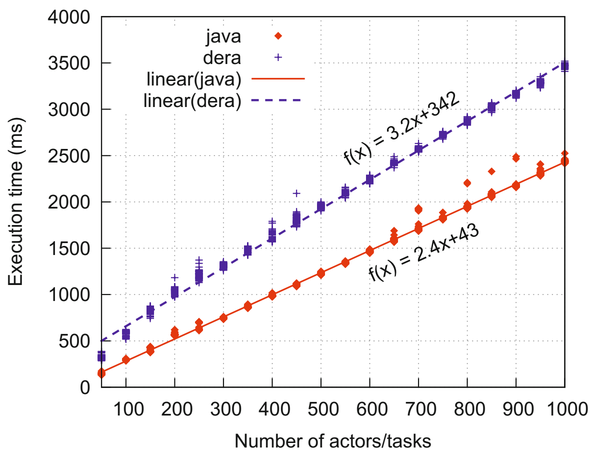
In the second requirement from Section 2 the warehouse operators need to call the warehouse staff to prepare for unloading products in the truck. This implies that a new actor, namely, `CallWarehouseStaff`, must be executed before the `StartUnloading` actor. We demonstrate in Listing 1.3 how the new actor can be incorporated into the existing warehouse operator composition logic using weak substitutions.

```
EventActor CallWarehouseStaff input[operatorStoreStartedNotified] output[
operatorCallWarehouseStaff]
EventActor StartUnloadingNew input[operatorCallWarehouseStaff] output[operatorStartUnloading]
register [CallWarehouseStaff]
register [StartUnloadingNew]
deactivate [StartUnloading]
/* verifications can be performed here to detect anomalies */
activate [StartUnloadingNew]
activate [CallWarehouseStaff]
```

Listing 1.3. Adding a new event actor `CallWarehouseStaff`

4.4 Performance and Scalability Evaluation

As the integration layer stands between the service-based applications and the underlying systems and platforms, it is a potential bottleneck in an integration architecture. Also the channel concept of DERA might introduce a bottleneck that could cause scalability problems. Thus, we conducted an evaluation of the performance and scalability of our approach comparing to a reference implementation based on pure hard-coded Java with rigid dependencies among the tasks. In this reference implementation we used exactly



tasks	$\overline{java}$	σ_{java}	$\overline{dera}$	σ_{dera}
50	148	8.2	323.9	23.7
100	293.9	5.1	582.4	26.4
150	403.5	22.1	819.0	31.1
200	569.9	12.7	1042.3	34.4
250	640.9	32.5	1216.0	46.9
300	745.5	6.6	1310.6	22.9
350	865.2	8.7	1484.4	28.8
400	986.4	7.6	1648.8	47.9
450	1103.5	8.6	1813.2	56.5
500	1224.9	8.8	1951.6	22.3
550	1342.1	8.0	2095.7	25.0
600	1466.5	11.2	2245.2	27.8
650	1592.4	20.0	2418.9	20.6
700	1734.8	77.0	2565.7	19.0
750	1827.1	15.6	2721.4	18.9
800	1954.8	52.3	2869.0	20.3
850	2076.6	38.8	3019.5	22.8
900	2186.4	60.8	3165.4	18.9
950	2309.9	25.5	3290.0	24.7
1000	2439.8	16.8	3464.6	22.1

Fig. 7. Evaluation of DERA scalability

the same tasks (a simple service invocation in a server running on the same machine) as in the DERA actor's tasks. We hard-coded the integration in Java, offering no flexibility, to exactly measure the impact of DERA's features on performance and scalability.

We evaluate the DERA implementation and the Java counterpart on the Java SE Runtime 1.6.0_u31 64-bit version on a workstation with an Intel CPU Quad-core i7 2.0 GHz and 8 gigabytes memory. To minimize the interference of the Java VM garbage collector and dynamic memory allocations during the experiments, the Java VM is set up with the following options: `-Xms512m -Xmx1024m -Xss1m`. We measured in 50 rounds the execution time of n ($n = 50, 100, \ldots, 950, 1000$, respectively) DERA actors running in an execution domain with a fixed thread pool of size 8 (which is the number of CPU cores) and compare to n Java tasks running in a thread pool of the same size.

Scatter plots for the measured execution times are visualized in Figure 7 (each value of n in the 50 rounds is depicted; the values are pretty close together). We derived the two regression functions shown in Figure 7 from the data of the measurements using the least-squares linear regression method. In Figure 7, we also present the average execution time of DERA (i.e., $\overline{dera}$) and the Java counterpart (i.e., $\overline{java}$) along with their standard deviations, σ_{dera} and σ_{java}, respectively. As can be seen for the observed data, both our approach and the Java hard-coded implementation offer approximately linear scalability. Our approach introduces only a moderate performance overhead, especially when considering that (1) realistic model sizes – such as those in our industrial case study – seldom go beyond 20-50 actors and (2) the approach is indented to be used for remote service integration and each service invoked over the network requires much more time than what is spent in the integration layer.

5 Related Work

The integration layer targeted in our approach is related to dynamic service composition approaches [10,2,7]. The dynamicity of those approaches is mostly achieved by deferring service discovery and binding to runtime. Initially, service placeholders or composition rules are prescribed in the configuration so that the enactment engine can later find, select, and combine relevant services on the fly. Moreover, most of these approaches using rigid dependency structures. However, none of the aforementioned approaches provides sufficient supports for flexibly changing or substituting arbitrary elements as in DERA. There are a few exceptions, such as the eFlow framework [6], in which modifications of composite services are allowed, but in an ad-hoc manner. As DERA actors need to interact with and incorporate various services, our approach can benefit from the advanced techniques in discovering and binding services to enhance the dynamicity of the interaction between DERA actors and corresponding services. None of these approaches considers the flexibility at runtime as in our approach.

An extensively used approach for specifying service composition are process-driven SOAs [15]. A typical process comprises a number of tasks and a control flow defining the execution order of these tasks. BPMN[6] and BPEL[7] are a widely used languages for describing processes. Unfortunately, they expose tight dependencies among service invocations with rigid control flows and structures. The enactment of BPMN or BPEL descriptions is usually determined at design time and very difficult to change at runtime. There are a substantial amount of efforts focusing on relaxing the rigid structures of process descriptions, and, therefore, enable a certain degree of flexibility of process execution [13,24,23]. These approaches mainly target long-running transactional systems and still suffer from the tight dependencies among the tasks. In contrast, there exists no such rigid control flows in DERA, only the *virtual* relationships among actors. Changing these virtual relationships can be straightforwardly achieved by altering actor interfaces using DERA substitution mechanisms. Our approach focuses on flexible short-running composition logic for integrating software systems and platforms. A number of approaches leverage the aspect-oriented programming paradigm for support process modifications by specifying and weaving additional modules, such as logging, auditing, and security, into the original composition logic [9,8]. However, the aspect-oriented approaches do not aim at loosening the dependency structures and provide limited supports for flexible changes at runtime.

A considerable amount of studies in the field of coordination theory are investigating methods and techniques aiming at separating computation from coordination [3] and making the interaction between components explicit in terms of coordination models and protocols. However, as far as components are still aware of these models and protocols which are stored in the components, their computation and interaction with peers is likely based on, and influenced by, this data [3]. In DERA, computational elements (i.e., actors) are totally unaware of the others. Communication between event actors is fully decoupled from the behavior of the actors.

The theoretical foundation of our work benefits from existing formal methods research in the field of subtyping. DERA substitution mechanisms are based on the studies on the substitutability of data types, objects, and components, especially the well-known Liskov substitution principle [17]. DERA extends these concepts to the domain of event-driven architectures and actors in behavior models. The implementation of

[6] `http://www.omg.org/spec/BPMN/2.0/PDF`
[7] `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`

DERA, in particular, the event channels, can be realized using asynchronous communication styles. Therefore, our future plan is to investigate and realize event channels using existing distributed publish-subscribe frameworks such as PADRES [11].

The concept of actor proposed in DERA is different from the *actor models* originally proposed in Agha's dissertation and follow-on studies [1]. The actors in the actor models are more complex as they encapsulate data, method, and interfaces. Moreover, in contrast to DERA systems in which event actors are totally unaware of each other, the actor models require that an actor must know the references, namely, *mail address*, of other actors to communicate with them by exchanging messages. This constraint clearly imposes tight dependencies among actors.

6 Conclusion

In this paper we present dynamic event actors (DERA) as a novel approach that exploits EDA to enable the flexibility of integration architectures and support various kinds of runtime evolution and adaptation. In particular, DERA introduces the concept of event actors with formally specified event interfaces for representing constituting elements (e.g., components, adapters, proxies). The communications and dependencies between actors are neither embedded in the actors nor prescribed in rigid dependency structures as in existing approaches. In contrast, the event-based communication style is exploited for loosening these dependencies among actors. In addition, event substitution mechanisms are proposed to enable the ability of altering DERA applications at runtime by making changes of event actor interfaces and substituting event actors. The main focus of our paper is to introduce DERA concepts and elements grounding on a sound formalization along with a prototypical implementation. The applicability of DERA has been shown through an industrial case study. The evaluation of DERA systems shows linear scalability with moderate performance overhead compared to an equivalent pure Java reference implementation.

A future work plan is to utilize existing formal reasoning methods for concurrent and distributed systems for supporting the verification of DERA system properties such as reachability, boundedness, and liveness, at important stages of development, e.g., before deploying and/or substituting actors. In addition, existing approaches for establishing reliable communication channels shall be exploited and extended in the context of DERA.

Acknowledgment. This work was partially supported by the European Union FP7 project INDENICA (`http://www.indenica.eu`), Grant No. 257483.

References

1. Agha, G.A.: ACTORS: A Model of Concurrent Computation in Distributed Systems. PhD thesis (1985)
2. Alamri, A., Eid, M., Saddik, A.E.: Classification of the state-of-the-art dynamic web services composition techniques. Int. J. Web Grid Serv. 2(2), 148–166 (2006)
3. Arbab, F., Talcott, C.L. (eds.): COORDINATION 2002. LNCS, vol. 2315. Springer, Heidelberg (2002)

4. Atluri, V., Chun, S.A., Mukkamala, R., Mazzoleni, P.: A decentralized execution model for inter-organizational workflows. Distrib. and Parallel Databases 22(1), 55–83 (2007)
5. Basten, T., van der Aalst, W.M.P.: Inheritance of behavior. Journal of Logic and Algebraic Programming 47(2), 47–145 (2001)
6. Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., Shan, M.-C.: Adaptive and Dynamic Service Composition in eFlow. In: Wangler, B., Bergman, L.D. (eds.) CAiSE 2000. LNCS, vol. 1789, pp. 13–31. Springer, Heidelberg (2000)
7. Chakraborty, D., Joshi, A.: Dynamic service composition: State-of-the-art and research directions. Tech. Rep. TR-CS-01-19, Department of Computer Science and Electrical Engineering, University of Maryland, USA (2001)
8. Charfi, A., Mezini, M.: AO4BPEL: An aspect-oriented extension to BPEL. World Wide Web 10(3), 309–344 (2007)
9. Cibrán, M.A., Verheecke, B., Vanderperren, W., Suvée, D., Jonckers, V.: Aspect-oriented programming for dynamic web service selection, integration and management. World Wide Web 10(3), 211–242 (2007)
10. D'Mello, D.A., Ananthanarayana, V.S., Salian, S.: A Review of Dynamic Web Service Composition Techniques. In: Meghanathan, N., Kaushik, B.K., Nagamalai, D. (eds.) CCSIT 2011, Part III. CCIS, vol. 133, pp. 85–97. Springer, Heidelberg (2011)
11. Fidler, E., Jacobsen, H.A., Li, G., Mankovski, S.: The PADRES distributed publish/subscribe system. In: Feature Interactions in Telecommunications and Software Systems VIII, ICFI 2005, pp. 12–30 (2005)
12. Ganesan, S., Yoon, Y., Jacobsen, H.A.: NIñOS take five: the management infrastructure for distributed event-driven workflows. In: 5th ACM Int'l Conf. on Distributed Event-based System (DEBS), pp. 195–206. ACM (2011)
13. Hallerbach, A., Bauer, T., Reichert, M.: Capturing variability in business process models: the provop approach. J. Softw. Maint. Evol. 22, 519–546 (2010)
14. Hens, P., Snoeck, M., Backer, M.D., Poels, G.: Transforming Standard Process Models to Decentralized Autonomous Entities. In: 5th SIKS/BENAIS Conf. on Enterprise Information Systems, pp. 97–106. CEUR WS.org, Aachen (2010)
15. Hentrich, C., Zdun, U.: Process-Driven SOA: Patterns for Aligning Business and IT. Infosys Press (2012)
16. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (April 1985)
17. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems 16(6), 1811–1841 (1994)
18. Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, Boston (2001)
19. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus, 1st edn. Cambridge University Press (June 1999)
20. Mühl, G., Fiege, L., Pietzuch, P.: Distributed Event-Based Systems, 1st edn. Springer (2006)
21. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE 77(4), 541–580 (1989)
22. Pierce, B.C.: Types and Programming Languages. The MIT Press (February 2002)
23. Redding, G., Dumas, M., ter Hofstede, A.H.M., Iordachescu, A.: Modelling flexible processes with business objects. In: IEEE Conf. Commerce and Enterprise Computing (CEC), pp. 41–48 (2009)
24. Reichert, M., Dadam, P.: Enabling adaptive process-aware information systems with ADEPT2. In: Handbook of Research on Business Process Modeling, pp. 173–203. Information Science Reference (2009)
25. Tombros, D., Geppert, A.: Building Extensible Workflow Systems Using an Event-Based Infrastructure. In: Wangler, B., Bergman, L.D. (eds.) CAiSE 2000. LNCS, vol. 1789, pp. 325–339. Springer, Heidelberg (2000)