Enterprise Integration Using Event Actor Based Event Transformations

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

ABSTRACT

Event-driven architectures can be used to enhance the flexibility of software system integration solutions by supporting flexible runtime changes of event processing rules, e.g., in complex event processing (CEP) engines. However, this often leads to integration solutions that are hard to maintain, complex, hard to reuse, and hard to understand. One reason are complex dependencies of events like high-level events mapped to low-level events or transformations of events between integrated systems (such as event aggregation, enriching, or splitting) that are hard to understand only by studying the interplay of various CEP rules. Another reason is tangled code spread across multiple artifacts including event processing code, event monitors, event listeners, event transformation code, existing system components that must raise or receive events. In this paper we propose to base integration architectures on event actors. Our approach consists of a model-driven event transformation framework that allows us to specify event actor based integration architectures as an architectural view (instead of tangled in the source code), as well as an event actor execution engine that supports the flexible deployment and enactment of the integration architecture. We show that a set of transformation-related enterprise integration patterns can be specified and flexibly enacted with our approach, the complexity is significantly reduced compared to purely CEP-based solutions, and a higher degree of reusability is supported.

1. INTRODUCTION

Software system integration refers to integration of independent systems that can run on their own but coordinate with each other in a loosely coupled way [12]. Various (distributed system) architectures have been proposed for enterprise integration (see [1, 4, 11, 12]), including messaging, service-oriented architectures, enterprise services buses, and many more. In all of them, enterprise integration concerns are not treated as first-class citizens from a software architectural perspective, but rather tangled across various integration-related software components. Also, many solutions require re-compilation, re-generation, and/or re-deployment to enact changes (see, e.g., [2, 21, 22, 25–27]). However, rapid changes are required in many systems, for instance because of frequently changing business rules or organizational changes that affect the enterprise integration, and not well supported by this approach.

Event-Driven Architecture (EDA) is an architectural style and approach for realizing applications and systems in which events are transmitted between decoupled software components and services [17]. EDA supports flexibility through decoupled interactions, many-to-many communications, event-based triggering of actions, and asynchronism [17]. This allows for supporting flexible runtime changes of event processing rules, for instance, by using complex event processing (CEP) engines [16, 19]. But this flexibility comes at the price of reduced understandability and higher complexity.

For example, service-oriented architecture is often used for enterprise integration and can be combined with EDA. Additional components and code for supporting EDA-based integration leads to more tangled integration code, as well as additional components that need to be understood, such as event monitors, event transformations, and event processors. In addition, the event-driven paradigm itself introduces a further level of complexity: Compared to message or invocation dependencies, event dependencies can be hard to understand, e.g., by only studying the interplay of various CEP rules, such as high-level events mapped to low-level events, or transformations of events between integrated systems (such as event aggregation, enriching, or splitting).

In summary, a major maintenance problems of integration architectures is addressed by introducing EDA for integration tasks (i.e., the missing flexibility), but other ones are worsened (higher complexity, reduced reusability).

In this paper we propose to base integration architectures on DERA, an event actors framework. Our approach encapsulates system integration components in stateless event actors with explicit interfaces. The approach exploits the event-based communication style to loosen the dependencies among actors. Also, the interfaces are formally specified and constrained to enable support for changing actors at runtime (e.g., replacing actors or changing their execution order).

We further propose to augment our event actor framework with a model-driven event transformation framework that allows us to specify event actor based enterprise integration architectures. That is, we provide meta-models for defining the integration architecture based on DERA concepts, such as event actors and events. The models derived from these meta-models can be used to generate all necessary components and source code artifacts needed for the implementation of the integration solution. This way, we can avoid the complexity and reduced reusability caused by tangled code (e.g., of CEP-based solutions).

The remainder of this paper is organized as follows: In Section 2 we discuss the enterprise integration architecture required for com-

SAC'13 March 18-22, 2013, Coimbra, Portugal.

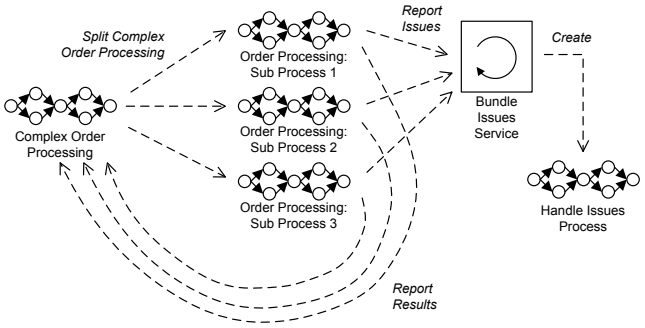

Figure 1: Handling Complex Orders Example: Enterprise Integration Architecture Sketch

plex order processing in an industrial case study as a motivating example. Next, we explain the concept to use event actors for supporting enterprise integration architectures in Section 3. Section 4 contains an overview of our approach, the details of the meta-models of our architectural view for event-based integration architectures and model-driven generation, and the integration with DERA. Section 5 illustrates how the example from Section 2 can be realized using our approach. Then we evaluate our approach with regard to complexity and reuse metrics in the context of the enterprise integration patterns in Section 6. Finally, in Section 7 we compare our work to the related work and conclude in Section 8.

2. MOTIVATING EXAMPLE

To illustrate the architectural enterprise integration challenges, let us consider an industrial case study originally described by Hentrich and Zdun [11, pages 296 ff.]. In this case from the telecom industry, complex orders by customers consist of various suborders that concern different business units of the company. Processing of suborders happens in parallel in different enterprise information systems to speed up order delivery. To improve customer satisfaction and reduce costs, issue with the order should be clarified with the customer before the goods are delivered. The customer's perspective is the whole order. Hence, the customer should not be individually contacted for each of the suborders.

As shown in Figure 1, enterprise integration must happen for splitting the arriving order into suborders and forwarding them to the suborder processing systems. Also, integration for aggregation of issues and results of the suborder processing is needed. A service, containing a bundle process agent, is used for aggregation of issues. If issues occur, a new process for handling the issues together is started.

In a typical process-based information system that uses messaging as an integration solution, usually redeployment is necessary for each change in the message routing, the routing rules, or the used integration patterns (like message enriching, aggregation, etc.). However, such changes may be frequent, as business organization and business rules frequently change. This is problematic, as most of the processes are long-running processes. Hence, complex solutions with runtime maintainable components that support starting and stopping of message queues for maintenance actions, such as redeployment of changed component, must be developed. It is the goal of our event actor based enterprise integration architecture to support changes more flexibly, e.g. by simply adding or replacing event actors.

3. EVENT ACTORS FOR SUPPORTING ENTERPRISE INTEGRATION ARCHITECTURES

The first part of our approach is to base enterprise integration architectures on an event actor framework. In this section, we discuss our event actor framework, called DERA (Dynamic Event-driven Actors), which exploits the event-based communication style to loosen the dependencies among actors in an integration architecture. The goal is to enable flexibility of integration architectures through runtime evolution and dynamic adaptations while at the same time minimizing the non-deterministic nature of event-based applications.

The central notions of DERA are *events*, *event actors*, and *event channels*. An event can be considered essentially as "any happening of interest that can be observed from within a computer" [19] (or a software system). Examples of events from our motivating example in Section 2 are: the placement of a complex order, the start of a sub-order process, and so on. An event might contain some attributes such as its unique identification, timing, data references, and so forth. We define DERA events and event types as follows:

Event type and instance - *An* event type *is a representation of a class of events that share a common set of attributes. An* event 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.*

Event actors are executing elements that have interfaces described in terms of incoming and outgoing events. The execution of an actor will be triggered by *any* of its input events. At the end of its execution, the event actor will emit *all* of its output events that, in turn, may trigger the executions of other event actors. From an architectural point of view, the event actor can hence be defined based on its interfaces (i.e., its externally visible properties):

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 finite set of input events expected by x and $x\bullet$ is a finite set of output events to be emitted by x ($\bullet x$ and $x\bullet$ can be empty sets).*

Actors are used in our enterprise integration approach for encapsulating software integration components. Well-defined interfaces for DERA event actors are important for our architectural integration approach, as they help to limit the non-deterministic nature of event-driven architectures. In particular, the well-defined actors interfaces enable us to derive a directed graph of the current architectural configuration comprising event actors connected via their inputs and output events. Therefore, we are able to monitor and analyze important properties, such as reachability (safety or deadlock checking), liveliness, performance, and quality of services of DERA systems. On the other hand, well-defined interfaces also enable us to perform changes at runtime, for instance, substituting an event actor by another with a compatible interface or changing the execution order of event actors by substituting an event actor with another.

The flexibility of DERA relies on the concept that each event actor only concentrates on its own task, as well as its incoming and outgoing events defined via its well-defined interfaces. There are no tight dependencies between two particular event actors, only the event-based communications. This is realized using the notion of *event channels*. All even actors in 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 of DERA. Two DERA channels can be connected by *event bridges* which are special event actors respon-

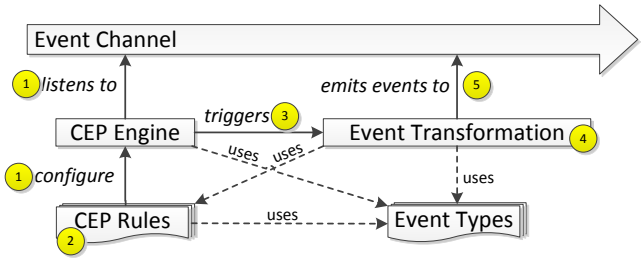

Figure 2: Architectural Setup of an Event Channel for Event Transformation

sible for forwarding events between two DERA event channels.

4. MODEL-DRIVEN EVENT-BASED INTEGRATION ARCHITECTURES

In this section, we propose a framework to create event-based integration architectures. Section 4.1 describes the architectural setup of an event channel for supporting the event transformations. Meta-models, described in Section 4.2, which are based on the DERA foundational concepts, defined in Section 3, facilitate the definition of architectural views for event-based enterprise integration in terms of event transformations in a DERA system. Based on the meta-models, we have defined model-to-code transformations for generating a DERA enterprise integration architecture from the models. For our implementation we used Esper[1] as CEP engine, the meta-models were realized with the Eclipse Modeling Tools[2], and for the model-to-code transformation XPand[3] was used.

Compared to simply using event rules, e.g. enacted in a CEP engine, our architectural view and model-driven transformation approach for DERA has many advantages: First of all, the architectural view enables us to model system integration concerns as an event transformation flow rather than using CEP rules. That is, realizing integration with rules on event patterns, e.g. with complex event processing (CEP) rules, happens on a rather low level of abstraction. A lot of detailed information about the system is needed and the integration rules have to be written manually, which is time consuming and error prone. Secondly, only CEP rules are not enough. For enacting the event-driven integration architecture, in addition, multiple artifacts, such as event processing code, event monitors, event listeners, event transformation code, existing system components that must raise or receive events, and so on, are needed. It is hard to understand all these elements and their interplay. Via the model-driven approach we can automate their generation. Finally, the event-driven actors architecture ensures the flexible changeability of the integration architecture – without sacrificing architectural understandability or control (see Section 3).

4.1 Architectural Setup

To illustrate how the model-driven approach reduces development complexity, let us consider the architectural setup of an event channel for supporting the event transformations, as illustrated in Figure 2. In our approach we use a CEP engine inside of an event actor to detect event patterns. Transformations are triggered by the detected patterns and enacted by event transformations deployed in actors. The following parts are needed:

[1]http://esper.codehaus.org/
[2]http://www.eclipse.org/modeling/emft/?project=ecoretools
[3]http://www.eclipse.org/modeling/m2t/?project=xpand

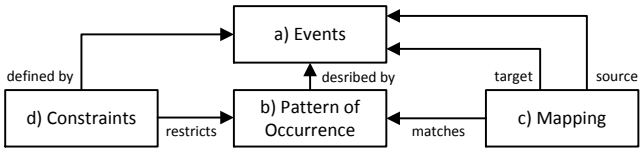

Figure 3: Overview of the Meta-Models and their Relations

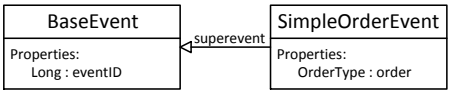

Figure 4: Example Event Type Model Instance

1. The CEP engine has to be set up to listen to the appropriate event channel, to recognize the correct event types and to load the rules describing the patterns to listen for. A special event actor listens to the channel and forwards events to the CEP engine.
2. The CEP patterns have to be described for each set of event occurrences. Also, variables are used to provide access to the events' properties.
3. If a pattern is detected, the event transformation and emission has to be triggered.
4. For each specified CEP pattern, the set of detected events has to be transformed to a set of output events. The transformations use the variables (see Part 2) to create the properties of the output events.
5. The output events created in Part 4 have to be emitted to the event channel. This is done using event actors for event transformations.

The benefit of our model-driven approach is that this architectural setup can be automatically generated via model-to-code transformations.

4.2 Meta-Models

To express event-based integration we developed the meta-models briefly described in this section. They mainly concern the event transformations that need to be defined to use DERA for event-based enterprise integration. The meta-models address four concerns: a) expressing event types, b) describing patterns of event-occurrence, c) mapping source patterns to target patterns, and d) expressing constraints for the mapping. The relation between these meta-models is depicted in Figure 3.

a) The first meta-model defines the DERA events and their event types. An instance of this model describes the set of available event types, which can be used by the actors of the DERA framework described in Section 3. An event can be modeled as a subtype of an event, and contains a set of properties. A property is defined by a name and a type. Figure 4 shows a sample event type model instance: the *SimpleOrderEvent* is a subtype of *BaseEvent*, containing the properties *order* and *eventID*.

b) The second meta-model specifies patterns of event occurrence. Whenever the emitted events on the event channel match the defined occurrence pattern, the event transformation of these source events will be triggered. To describe the occurrence patterns event containers are used to group a set of events. A container may reference a single event, express set operations (e.g., *and*, *or*) or a sequence, and contain further containers. Figure 5 shows an example for nested event containers, expressing the occurrence of either an event of the type `E3`, or two events of the type `E1` and

E2. Also, additional constraints can be defined for a set of events, which is described in more detail below.

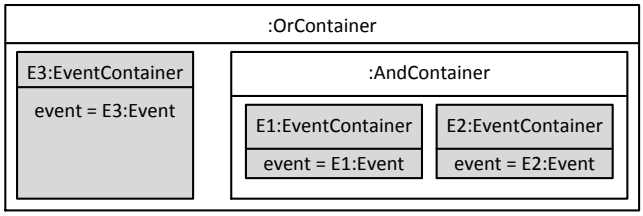

Figure 5: Nested Event Container Example:E3 or (E1 and E2)

c) The third meta-model specifies the mapping of source events to target events. It is used to describe the transformation of events matching occurrence patterns into a set of target events during runtime. For every expected target event, it is described which source events and which properties of the source events have to be transformed into the target event. Variables and code injectors are used to provide additional data which is not provided by the source events. In Figure 7, the *:ResultEventMapping* provides a mapping from a ComplexOrderEvent to a SimpleOrderEvent.

d) Finally, the fourth meta-model allows us to define event transformations using constraints for events in occurrence patterns. For example, a constraint can describe that a transformation only has to be performed, if a specific property of an event has a well-defined value. For instance, the following constraint will restrict the matching of an occurrence pattern to only the event `E1` when its property `p1` has the value "1" or "2": `E1.p1 == '1' OR E1.p1 == '2'`

Instances of the meta-models are used to generate artifacts to enact event transformations using model to code transformations. The generated artifacts are Java code fragments representing DERA concepts such as events and actors, as well as configurations and rules for the CEP engine Esper. Our transformation generates the following artifacts to enact the modeled event transformation:

Event Types: Instances of the Event meta-model are transformed to Java classes implementing a DERA Event as described in Section 3.

Esper Rules: Esper Statements represent the patterns of occurrence in a textual form, created from Container and Constraint instances of the Event Transformation meta-model.

Pattern Detection Listener: For every Esper rule, a listener is generated. It is notified if Esper notices the occurrence of an event pattern in the event channel. A listener is responsible for triggering the event transformation and the target event emission.

Transformation: Based on instances of *ResultEventMapping*, the code for event transformation is created. Our framework provides support for type conversion for primitive types (e.g., integer to float) and also complex types (e.g., string to integer).

Esper Setup: A configuration for setting up the Esper engine is generated using information from instances of the Event and Event Transformation meta-models. This configuration is responsible for registering the event types to be detected by the Esper engine and to register the Esper rules and its Pattern Detection Listeners.

5. MOTIVATING EXAMPLE RESOLVED

To implement the motivating example described in Section 2, we designed a DERA application with two channels, depicted in Figure 6. The source channel consists of two DERA actors *SendComplexOrderActor* and *ResultHandlingActor*, where the first is responsible for emitting an event containing a set of orders, and the latter waits for the results of each atomic order processing. The *SendComplexOrderActor* actor is able to emit an event of type *ComplexOrderEvent*, which consists a set of orders. The *ResultHandlingActor* is listening for an event of type *HandleResultsEvent*, which contains all information about the processed orders. The target channel consists of two actors, a *SubOrderProcessor* and an *IssueProcessor*. The *SubOrderProcessor* is triggered by an instance of a *SimpleOrderEvent* event. When it finished processing the order, it sends out two events of the types *OrderResultEvent* and *OrderIssueEvent*. The first contains information of a successful proceeded order, the latter contains an error report (if an error occurred). The *IssueProcessor* is triggered by events of type *HandleIssueEvent*.

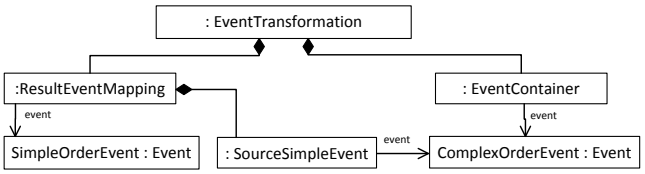

Figure 7: Schematic Transformation Model Instance of the Splitter

The *ComplexOrderEvent* contains a collection of orders specifications, whereas a *SimpleOrderEvent* only contains a single order specification. We modeled two event transformations: one splitting the *ComplexOrderEvent* into *SimpleOrderEvents* for each containing order, and one aggregating the *OrderIssueEvents*.

Figure 7 shows the schematic transformation model instance to split up *ComplexOrderEvents* into *SimpleOrderEvents*. The *EventTransformation* consists of an *EventContainer* and a *ResultEventMapping*. The *EventContainer* listens to every event of the type *ComplexOrderEvent*. The *ResultEventMapping* emits events of the type *SimpleOrderEvent* that have to be created from values of the *ComplexOrderEvents* defined by the *SourceSimpleEvent* element. This model instance contains all information which are necessary to automatically create the artifacts described in Section 4 and 4.2. First, an Esper rule is created, representing the pattern of the *EventContainer*. In the example, a simple Esper rule is created: `every( ComplexOrderEvent=ComplexOrderEvent )`. This means, that the transformation has to be triggered on every occurrence of event of the type *ComplexOrderEvent*. Also, the event is named by its type name, so the event is accessible through the Esper engine during runtime. Second, a pattern detection listener is generated. This listener is triggered, when the Esper rule is detected on the source channel. Also, the listener is able to execute the transformation from the source event into the target event. In our example, for every order in the ComplexOrderEvent a SimpleOrderEvent is created and emitted on the target channel. Third, an Esper setup is created which contains the configuration of the Esper engine. For instance, the Esper rule is registered at the Esper engine and connected to the pattern detection listener. Also, the event types to be listened to are registered at the Esper engine.

The presented example illustrates that our model-based definition of an architectural view for enterprise integration reduces the complexity compared to purely CEP-based solutions. Only one clear model instance, which is easy to understand, is needed to generate all necessary artifacts. Therefore, there is no need to manually manage tangled code spread across several artifacts.

6. EVALUATION

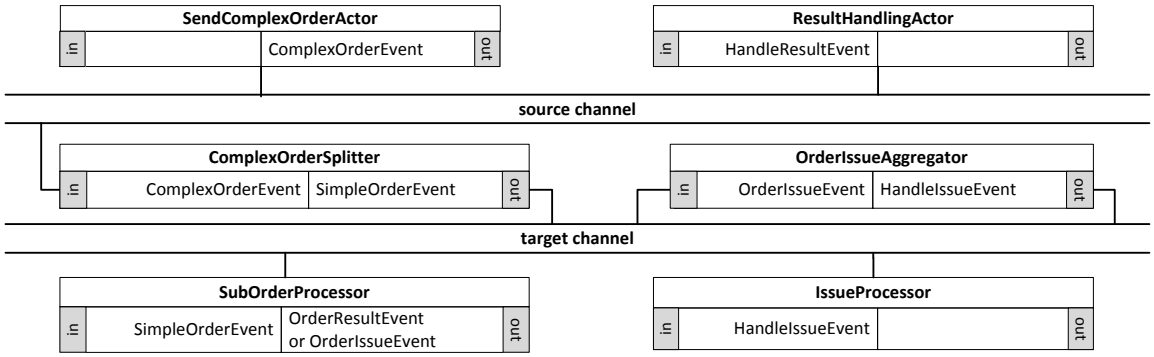

Figure 6: Overview of the DERA Solution for the Motivating Example

We show that our approach significantly reduces complexity compared to purely CEP-based solutions and a higher degree of reusability is supported. The feasibility of our approach is shown by implementing a sub-set of the Enterprise Integration Patterns by Hohpe and Woolf [12]. In particular, as our approach is mainly focused on event transformation, we studied the whole pattern language by Hohpe and Woolf and selected all those patterns that have a clear (message) transformation focus.

We used a metrics-based quantitative analysis to show that our approach decreases the complexity in system integration scenarios corresponding to the selected integration patterns. A frequent used metric for quantifying complexity of software still is Lines of Code (LOC) [10], where the lines of a source code except blank lines and comments are counted. As models are not organized in sequences of characters, the LOC metric is not suitable for comparing models. Lange [15] suggests to use the number of model elements to express the absolute size of a model. To compare the complexity of our models with the CEP-based solution implemented in Java and Esper, we converted the Java/Esper textual code into corresponding models. To analyze Java artifacts, we used the Eclipse Java Development Tools (JDT) implementation of the Java model. We counted elements of the following types: Statement, BodyDeclaration, ImportDeclaration, PackageDeclaration, VariableDeclaration, and TypeDeclaration. All other types (e.g., Comment or Annotation) were ignored. The Esper rules were expressed using the Esper Statement Object Model.

The result of the comparison between the model elements and the source code elements is shown in Table 1. The first column shows the name of the integration pattern modeled in the scenario. The second column shows the number of the model elements E_m needed to express a pattern. The next three columns show the number of elements E_s counted in the Java and Esper models of the CEP-based solution, as well as the sum of them. By using the Java/Esper code generated from our models for the comparison we made sure that we compared exactly the same scenarios. The last column shows the complexity ratio CR between the model elements and the sum of source code elements, where $CR = E_m/E_s$. Note that the comparison of the model elements are on a statement granularity. Because of an Esper Pattern is a part of an Esper Statement, even very complex Esper Patterns are expressed by only 1 Esper Statement.

The complexity ratio shows that the effort to model an integration pattern scenario is in average about 7.45 percent lower compared to the code needed for creating the artifacts manually using the CEP-based solution. That is, there are about 13.4 times (= 1 / 7.45 %) less model elements needed in average in our solution than in the CEP-based solution to express an integration pattern scenario. In addition, the numbers show that the CEP-solution requires substantial Java coding. Hence, from an architectural point of view, we can assess that the CEP-based solution is substantially more low-level than our approach.

Table 1: Comparison of Complexity of Integration Pattern Scenarios

Pattern Name	E_m	E_s			CR
		Java	Esper	sum	(%)
Aggregator	37	377	1	378	9.79
ContentBasedRouter	33	281	1	282	11.70
ContentEnricher	15	285	1	286	5.24
ContentFilter	29	346	1	346	8.38
MessageRouter	14	302	2	305	4.59
MessageTranslator	15	274	1	275	5.45
Normalizer	29	412	2	414	7.00
Splitter	21	350	1	351	5.98
WireTap	11	123	1	123	8.94
Average					7.45

Note also that we only analyzed the artifacts directly derived from the models, which are based on a framework with 6470 Java model elements. The number of elements will increase considerably if the Java implementation for only one pattern will be done from scratch without this framework.

We also analyzed the reusability of our model elements. The IEEE Standard Glossary of Software Engineering Terminology [13] defines reusability as: *"The degree to which a software module or other work product can be used in more than one computer program or software system"*. In the context of model-driven development (MDD), Tran [25] refines this definition as: *"the degree to which a model can be used in more than one software or system. As such, the reusability of a model can be measured by the amount of reusable model elements and the model itself which can be used in more than one software or system without any changes or with small adaptations"*. We express the reusability by expressing the ratio of the model elements for describing an integration patter to the elements occurring in the generated artifacts. Table 2 shows the reusability of our model elements expressed by reuse coefficient R_c, which is defined as follows: R_c = R_s / E_m. The available amount of model elements is expressed by E_m. The amount of occurrence of these model elements within the source code artifacts (Java code and Esper rules) is expressed by R_s. The results show that every model element is reused 3.73 times in average.

The results of our evaluation show that our approach to model

Table 2: Reusability of Model Elements in Integration Pattern Scenarios

Pattern Name	E_m	R_s	R_c
Aggregator	37	109	2.95
ContentBasedRouter	33	74	2.24
ContentEnricher	15	77	5.13
ContentFilter	29	89	3.07
MessageRouter	14	71	5.07
MessageTranslator	15	72	4.80
Normalizer	29	113	3.90
Splitter	21	97	4.62
WireTap	11	20	1.81
Average			3.73

the architectural view on event transformation decreases the complexity of dependencies considerably compared to the tangled code of pure CEP-based solutions. Also, the high degree of reusability of our models in typical enterprise integration scenarios is shown in our evaluation.

7. RELATED WORK

Hohpe and Woolf propose the Enterprise Application Integration (EAI) patterns [12] as solutions to recurring integration problems, especially focusing on a perspective driven by messaging technologies. We selected transformation-related EAI patterns to show that those patterns can be realized, flexibly enacted, and reused with our approach. Scheibler et al. introduce executable parametrized EAI patterns [21] to support architects in the reuse of EAI patterns in software solutions and to parametrize the solution to fit to specific integration problems. Scheibler et al. use a model-driven approach to generate a workflow and service based implementation. They further propose to offer the executable parametrized EAI patterns as a service [22]. Like our approach, the approach by Scheibler et al. provides reusability of elements of integration architectures. However, in contrast to our event actor based approach, the approach by Scheibler et al. requires model-driven re-generation for enacting changes, as it uses workflows for the control flow of the EAI pattern composition.

Mendling et al. [18] present a view integration approach inspired by the idea of schema integration in database design, based on Event-Driven Process Chains (EPCs). The predefined semantic relationships between model elements of EPCs, – such as equivalent, sequence, and merge operations – were also investigated by Davis [7] and Kindler [14], who performed these operations to integrate two distinct views. Since it is hard to apply semantic-based merging in order to integrate two different types of models (e.g., merging a control flow model with a data model), the authors mainly focus on merging control flows. In contrast to those approaches, our approach rather aims at providing a flexible solution for software system integration than on merging processes. We used architectural views mainly for the purpose of reducing complexity and enhancing reusability of integration architectures, whereas the process view integration approaches use views mainly to integrate different process designs into a coherent process design.

Axenath et al. [2] propose a meta-model for formalizing different aspects of business processes called AMFIBIA. They provide an open framework to integrate various formalisms through a central notion of interface. The models in AMFIBIA roughly correspond to process-centric views. An approach focusing different levels of abstraction for business processes, for instance, abstract and technology-specific layers, is the View-based Modeling Framework (VbMF) presented by Tran et al. [25–27]. VbMF exploits the notion of views to separate the various process concerns of a business process to reduce the complexity of process-driven SOA development and enhance the flexibility and extensibility of the framework. In contrast to our event actors based approach, these approaches do not provide additional flexibility for integration solution. Rather they require model-driven re-generation of the source code to support changes in software integration solutions.

Our proposed solution is based on meta-models for describing event transformations. Therefore, one part of the meta-models have to describe the structure of events. Early event models only provided parametrized primitive events [3, 5, 6], which mainly lack in expressing event types. Type-based events were introduced by Eugster [8, 9] with the goal to make events first-class citizens in object-oriented programming languages.

Based on this concept, Event Stream Processing (ESP) and Complex Event Processing (CEP) systems [16, 19] drove the development of event models. Rozsnyai et al. [20] discuss existing event models and based on this research they propose an event model of an event-based system called SARI, focusing on representing, structuring and typing event data. They discuss concepts of existing event-based solutions and introduce basic typing concepts for structuring event data as well as more advanced typing concepts such as inheritance, exheritance, and dynamic type inferencing. In our work, we have used these approaches as a foundation, but tried to keep our event meta-model as simple as possible to reduce the complexity of our proposed event transformation meta-model, which is part of our main contribution.

Taher et al. [23, 24] proposed a framework to create CEP-based adaptors to connect Web Services, based on a set of five predefined operators to describe mappings between messages. From of these mappings, adapters can be generated capable of intercepting services and to modify structure, type, and number of incoming messages to output messages. This approach provides a solution for incompatibilities between signatures (interfaces) or protocols, but does not focus on incompatibilities between patterns of events which are essential for integrating event driven architectures.

8. CONCLUSIONS

In this paper we have proposed a novel approach for modelling and enacting enterprise integration architectures that combines concepts from event-driven architectures, model-driven development, and architectural views. We were able to combine the flexibility of event actor based architectures with an explicit architectural view for defining the integration architecture. This way we avoid the tangled code in various systems components, e.g. known from similar approaches such as integration architectures based on pure CEP-based solutions. In our evaluation we were able to show that we can realize the set of transformation-related EAI patterns using our approach, and that our approach has a positive impact on enhancing the reusability and reducing the complexity of integration architectures compared to pure CEP-based solutions.

The major limitation of our approach so far is that we mainly concentrated on transformation-related EAI patterns (explained in Section 6). As discussed in Section 7 many other kinds of integration-related patterns exist that can potentially be expressed using our approach. We plan to address this by extending our catalogue of reusable integration solutions with those other patterns in our future work and develop a repository of reusable integration fragments.

Another limitation of our study reported in this paper is that the evaluation is based on two metrics for measuring complexity and

reuse. While these metrics are congruent with our intuition regarding those properties, metrics are always limited in their validity. For example, the two metrics do not address all aspects of complexity and reuse, and our interpretation and comparison might be biased. Finally, so far our architectural view approach has not been integrated with other architectural view models. We do not see this as a big limitation, though, as the view models in our approach have been developed in close correspondence to other model-driven view models, such as VbMF [25–27] and we are hence confident that view-model integration is possible.

As also discussed in Section 7 will enable us to support semi-automatic or automatic optimization and adaptation of event-based integration architectures, for instance to optimize performance. We plan to address optimization and adaptation of event-based integration architectures in our future work.

9. REFERENCES

[1] P. Avgeriou and U. Zdun. Architectural Patterns Revisited – A Pattern Language. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*, pages 1–39, Irsee, Germany, July 2005.

[2] B. Axenath, E. Kindler, and V. Rubin. AMFIBIA: a meta-model for integrating business process modelling aspects. *International Journal of Business Process Integration and Management*, 2(2):120 – 131, 2007.

[3] G. Banavar, M. Kaplan, R. E. Strom, D. C. Sturman, K. Shaw, and W. Tao. Information flow based event distribution middleware. *Distributed Computing Systems, International Conference on*, 0:0114, 1999.

[4] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture – On Patterns and Pattern Languages*. Wiley Series on Software Design Patterns. John Wiley & Sons Ltd., Chichester, England, April 2007.

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

[6] I. Corporation. Gryphon: Publish/subscribe over public networks. Technical report, IBM T. J. Watson Research Center, 2001.

[7] R. Davis. *Business process modelling with ARIS: a practical guide*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[8] P. Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.*, 29(1), Jan. 2007.

[9] P. T. Eugster and R. Guerraoui. Content-based publish/subscribe with structural reflection. In *Proceedings of the 6th conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 6*, COOTS'01, pages 10–10, Berkeley, CA, USA, 2001. USENIX Association.

[10] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach (2nd Edition)*. PWS, 1998.

[11] C. Hentrich and U. Zdun. *Process-Driven SOA: Patterns for Aligning Business and IT*. Infosys Press, 2012.

[12] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Boston, MA, USA, 2003.

[13] IEEE. IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology, Dec. 1990.

[14] E. Kindler. On the semantics of EPCs: A framework for resolving the vicious circle. In M. Nüttgens and F. J. Rump, editors, *EPK*, pages 7–18. GI-Arbeitskreis Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, 2003.

[15] C. Lange. Model size matters. In T. Kühne, editor, *Models in Software Engineering*, volume 4364 of *Lecture Notes in Computer Science*, pages 211–216. Springer Berlin / Heidelberg, 2007.

[16] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[17] J.-L. Marechaux. Combining Service-Oriented Architecture and Event-Driven Architecture using an Enterprise Service Bus, 2006.

[18] J. Mendling and C. Simon. Business process design by view integration. In *Proceedings of the 2006 international conference on Business Process Management Workshops*, BPM'06, pages 55–64, Berlin, Heidelberg, 2006. Springer-Verlag.

[19] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer, 1 edition, 2006.

[20] S. Rozsnyai, J. Schiefer, and A. Schatten. Concepts and models for typing events for event-based systems. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, DEBS '07, pages 62–70, New York, NY, USA, 2007. ACM.

[21] T. Scheibler and F. Leymann. A Framework for Executable Enterprise Application Integration Patterns. In K. Mertins, R. Ruggaber, K. Popplewell, and X. Xu, editors, *Enterprise Interoperability III*, pages 485–497. Springer London, 2008.

[22] T. Scheibler, R. Mietzner, and F. Leymann. EAI as a Service - Combining the Power of Executable EAI Patterns and SaaS. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, EDOC '08, pages 107–116, Washington, DC, USA, 2008. IEEE Computer Society.

[23] Y. Taher, M.-C. Fauvet, M. Dumas, and D. Benslimane. Using cep technology to adapt messages exchanged by web services. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 1231–1232, New York, NY, USA, 2008. ACM.

[24] Y. Taher, M. Parkin, M. Papazoglou, and W.-J. van den Heuvel. Adaptation of web service interactions using complex event processing patterns. In G. Kappel, Z. Maamar, and H. Motahari-Nezhad, editors, *Service-Oriented Computing*, volume 7084 of *Lecture Notes in Computer Science*, pages 601–609. Springer Berlin / Heidelberg, 2011.

[25] H. Tran. *View-based and Model-driven Approach for Process-driven, Service-Oriented Architectures*. PhD in software engineering, Vienna University of Technology, Distributed Systems Group, Information Systems Institute, Argentinier Str. 8/184-1, Vienna A-1040, Austria, Dec. 2009.

[26] H. Tran, T. Holmes, U. Zdun, and S. Dustdar. *Modeling Process-Driven SOAs – a View-Based Approach*, chapter 2. Information Science Reference, Handbook of Research on Business Process Modeling edition, Apr. 2009.

[27] H. Tran, U. Zdun, and S. Dustdar. Vbtrace: Using view-based and model-driven development to support traceability in process-driven soas. *Software & System Modeling*, 2009.