Using Policies for Handling Complexity of Event-Driven Architectures

Tobias Freudenreich, Stefan Appel, Sebastian Frischbier, and Alejandro P. Buchmann

Databases and Distributed Systems, TU Darmstadt
lastname@dvs.tu-darmstadt.de

Abstract. Cyber-physical systems and the Internet of Things illustrate the proliferation of sensors. The full potential of ubiquitous sensors can only be realized, if sensors and traditional data sources are integrated into one system. This leads to large, complex systems which are harder to use and manage, and where maintaining desired behavior is increasingly difficult. We suggest a novel approach to handle the complexity of these systems: users focus on the desired behavior of the system and use a declarative policy language (DPL) to state these behaviors. An enhanced message-oriented middleware processes the policies and automatically generates components which execute the policies. We compared our approach against other approaches in a case study and found that it does indeed simplify the use of cyber-physical systems.

1 Introduction

The number of sensors in today's environments increases steadily. In our homes the sheer number but also the different kinds of sensors have increased in recent years. Similarly, companies rely more and more on sensor data to improve and steer their processes, especially in production and logistics. Trends like cyber-physical systems (CPS) or the Internet of Things illustrate this evolution further.

This calls for a new perspective on software architecture. In Event Driven Architectures (EDAs) components get triggered by events [27]. In service Oriented Architectures (SOAs) they are invoked by explicit calls. In modern architectures both interaction paradigms coexist. This perspective shift comes at the price of increased architectural complexity: EDAs are inherently distributed, the application does not have direct control over the control flow, and language-support for event processing is but well-supported. We believe that this complexity must -at least in part- be handled by a middleware, which abstracts from this complexity.

Event handling today is done via complex event processing (CEP), which proposes to construct more complex events out of simple ones, according to a set of rules [24]. An event query language (EQL), as for example found in Esper or Software AG's Apama, allows for declaratively stating such rules. However, this abstraction is still on a very low level. It requires expert developers to be handled correctly. Even when handled by experts, the number of rules quickly

exceeds a manageable amount, creating a maintenance nightmare with (hidden) dependencies among the rules and no indication why a rule was created.

Our goal is to allow domain experts without a profound computer science background to use systems based on an EDA, as well as to provide developers with a mechanism which abstracts from the architectural complexity of EDAs. Thus, we introduce the abstraction of a *policy*. A policy is the conceptual representation of a set of fine grained rules, which cooperate towards a common goal. Policies abstract from rule management and architectural distribution. By allowing users to state policies declaratively, we enable them to focus on *what* they want to achieve, rather than *how* this is done exactly. Other approaches, sharing the goal of simplifying the use of EBS by providing higher level abstractions, focus on procedural workflows [2, 19]. Declarative statements bear the advantage of abstracting from implementation details, while procedural statements provide more control of how a task is achieved. No concept is inherently superior.

We illustrate the idea of our approach with a typical example from the domain of situation monitoring originally proposed by Georgakopoulos et al. [18]: A company wants to ensure the following: guests are allowed to walk around the company area. However, in some restricted rooms, guests need to be accompanied by an employee, otherwise an alarm should sound. There are already many sensors in place to derive positions of people, e.g., RFID readers or cameras. [1]

Ensuring this policy involves several steps (querying the database for a person's status, calculating absolute positions, checking in which room a person is, correlating multiple position events). Each of these steps requires a number of low-level CEP rules to encode. A common approach to reduce the number of rules is to define event compositions, resulting in a more complex event hierarchy. When adding more "real-world" requirements, like different security clearances, the set of rules quickly becomes hard to manage. We therefore advocate a segmentation of the rule space into policies and a mechanism to express them declaratively[2]:

```
IF
  person A with attribute status='guest' IS INSIDE
  room R with attribute security='restricted' AND
  person B with attribute status='employee'
    IS NOT WITHIN 5m of A
THEN
  sound alarm
```

We provide a generic middleware architecture and approach to automatically process such policies. It can be instantiated by providing a domain model and annotating data sources with metadata. Similar to a database expert creating a database schema, providing this information is done only once. With this information, we can generate *Event Enrichment Components (EECs)* which enrich

[1] We were able to verify the validity of such a scenario in discussions with Software AG (http://www.softwareag.com), a leading provider of business application software.

[2] Note that this policy does not enforce B to be in the same room as A. We omit this detail for presentation simplicity.

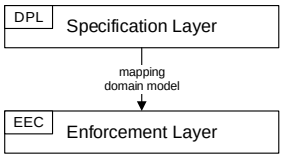

Fig. 1. Policies are specified in DPL and mapped to executable code in EECs

events with additional knowledge. EECs enforce policies by evaluating derived rules against incoming events.

Enabling domain experts to fully exploit CPS requires the following components (Figure 1 provides an illustration): a) A user-friendly way of stating the policies. We refer to this as the declarative policy language (DPL). b) A grammar underlying DPL. c) A mapping from DPL to executable code (EEC). d) A framework to capture domain concepts to support the mapping.

The contributions of this paper are:

- A novel approach to handle the complexity of cyber-physical systems and event-driven architectures in general.
- A generic, declarative language to state policies and a middleware architecture for processing them. The language can be mapped to multiple concrete languages and implementations.
- An implementation with modern message-oriented middleware to map DPL statements to executable code. Our implementation is in Java and based on Apache ActiveMQ, an industry-strength middleware.
- An approach that preserves the benefits of event-driven architectures: new sensors/producers can be added at runtime and the system remains flexible

The rest of this paper is structured as follows. Section 2 provides a detailed description of the parts comprising our DPL-enabled middleware, including how they interact. Section 3 compares our approach against Java and CEP-techniques in a case study. We discuss related work in Section 4 and conclude in Section 5.

2 Technical Overview

In this section, we detail how to get from a policy to executable code. Therefore, we first provide our domain model framework and the grammar for DPL. We follow up with the description of the architecture and close with an illustration of the interaction among the components with the example from Section 1

2.1 Domain Model Framework

We need domain models as background knowledge to generate executable code from a policy. The purpose of this section is not to discuss suitable representation languages, but rather state the requirements of our approach. We therefore introduce the individual elements and defer discussion about a suitable representation to Section 2.3.

Concepts and Attributes Every policy will refer to certain *things*. In the example from Section 1 these are `person` and `room`. In that regard they are similar to entities in a database schema. However, we chose to call them *concepts* (similar to description logic) to avoid confusion. Concepts are described by attributes and may not have a direct representation (i.e. in a database). They might exist implicitly only (e.g., because various events refer to it)

Relationships Relationships connect concepts. A relationship is backed by a relationship function, which evaluates for given instances if the relationship holds. A relationship function uses attributes of the related concepts. More than one relationship may exist between two concepts. For example, the employee and room can be connected by the relationship works in and meets customers in.

Figure 2 shows an example illustrating the concepts, attributes and relationships of our running example. The relationship *inside* refers to the attributes *position* and *coordinates* of two opposing corners of a rectangular room. Thus, when evaluating if a specific person is inside a specific room, the relationship function will be passed the values of position and coordinates as its arguments.

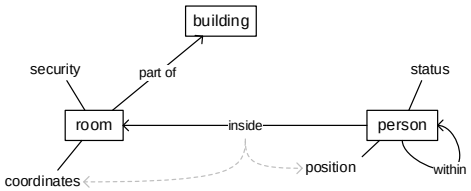

Fig. 2. Example domain model related to the policy from Section 1

To support relationships like *within*, which need an additional parameter (e.g., 5 meters), the framework must support parameterized functions.

Since relationship functions can be complex, we chose to keep them separate from the structural definition to support *separation of concerns*. In our example, one concern is to say that generally, persons can be inside rooms. Another concern is to say what exactly the semantics of being inside a room is.

Another advantage of keeping the definition of the relationship functions separate is reusability. Developers can reuse their function in different domain models and a set of functions can be directly shipped with the middleware.

2.2 Policy Grammar

With usability in mind, we believe that the determinism of a formal language outweighs the familiarity of natural language. Thus, policies must follow a formal syntax. Figure 3 shows the grammar that generates DPL in EBNF notation. The nonterminals concept, attribute, value, f-name, parameter and action are not explicitly given, as they can be arbitrary strings.

```
policy      ::= 'IF' conditions 'THEN' actions
conditions  ::= condition | conditions op conditions | '(' conditions ')'
condition   ::= concept-def function concept-ref
concept-def ::= concept [alias],
   ['with attribute' (attribute attr-op value)*]
attr-op     ::= '=' | '!=' | '<' | '>' | '<=' | '>='
concept-ref ::= concept-def | alias
function    ::= ['is'] ['NOT'] f-name [parameter] ['of']
op          ::= 'AND' | 'OR'
actions     ::= action+
```

Fig. 3. Grammar for policies in Extended Backus-Naur Form (EBNF)

Policies are divided into a situation description part and an actions part. The first part contains a set of conditions, describing the desired situation. Conditions are checked upon arrival of relevant events and, if met, the actions of the actions part are triggered. Thus, policies are similar to event-condition-action rules [10]. However, policies abstract from the notion of events and let users think in the descriptions of situations.

Concepts may have an alias for easier referencing within the policy. The `with attribute` statement allows for filtering out instances of a concept not relevant for the current statement. For example, guests and employees are persons, but we are concerned about guests being alone in restricted areas. Functions connect two concepts to a condition and may have a parameter, as indicated in Section 2.1. The element 'of' is only syntactic sugar: it makes policies more readable.

DPL is a generic language, which is instantiated by pairing it with a concrete domain model. The resulting language is specific to the given domain, but may map to several concrete programming languages like Java or C#.

2.3 Middleware Architecture

In event-driven architectures (EDAs), events (e.g., sensor readings) are reified as event notifications. Due to a typically high volume of data and the benefits of decoupling, EDAs adopt the publish/subscribe messaging paradigm [14]: software components with an interest in events issue subscriptions for them with a message-oriented middleware (MOM). Sensors act as event producers and send event notifications in the form of messages to the MOM. There, a message broker matches incoming messages against issued subscriptions and routes messages to interested consumers.

Figure 4 shows the generic architecture of the middleware enabling declarative behavior specification. We will detail the components in this section and illustrate how they interact. Components are loosely coupled and communicate asynchronously through a message bus. Interaction with the request/reply parts of the EDA also happen through messages (e.g., an event-driven SOA, where the action part of a policy fires a service-triggering event).

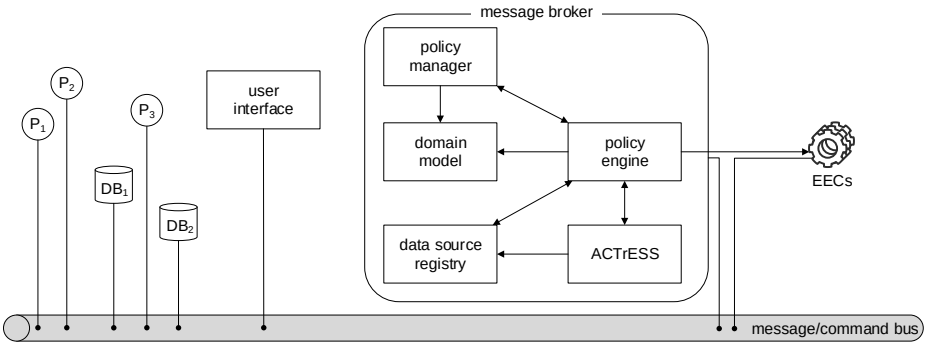

Fig. 4. Architecture of the DPL-enabled middleware

Message and Command Bus A message bus connects event producers, auxiliary data sources (e.g., databases), the user interface, the policy engine as well as the processing components. We use the existing messaging functionality of the MOM. The command bus consists of special message channels. Thus, each component is able to listen to commands asynchronously, which provides for good decoupling and easy distribution.

Producers send events via the publish/subscribe messaging facility. Since all messages pass through the broker, it can access message content.

User Interface The user interface allows for creating, altering and deleting policies. Creating a policy just requires writing it, choosing a unique name and sending the policy to the middleware. We provide users with an Eclipse plugin with content assist, syntax highlighting and syntax checking. Thus, users cannot accidentally send erroneous policies to the middleware. Altering a policy follows similar steps, except that the user first retrieves the policy by its name. Deletion of a policy may happen implicitly or explicitly. An explicit deletion means the termination by a user action, while implicit deletion happens as part of a policy's action part.

Policy Engine The policy engine is the heart of our DPL-enabled middleware. It serves as a coordinating component in a controller-like fashion: Upon receiving a policy, the policy engine uses the policy manager to analyze the policy. Based on the analysis, it generates *Event Enrichment Components* (EECs).

Conflicting Policies Especially in multi-user environments policies might conflict. For example, one policy could state to close the windows if the temperature is less than 25°C (to preserve heat), while another policy requires opening the windows if the temperature is greater than 23°C (to preserve air quality).

Without proper semantic annotation of actions, it is impossible to automatically analyze, on a semantic level, what the *effect* of an action is. Such semantic

analysis is out of scope of this paper and subject of future work. Currently, we support users by displaying *similar* policies to them as they edit theirs. Similarity is chosen based on the concepts and their attributes a policy is referring to.

Domain Model The policy engine uses the domain model when generating the EECs. We chose a custom representation in favor of more elaborate languages like OWL or RDF. The main reason for our choice is performance. Since we integrate various, different, already existing data sources, their information is not yet in the domain model. However, ontological reasoning requires this information to be in the model. Inserting on demand, then reasoning severely impacts performance. Alternatively, one could try keeping the model synchronized with external data sources, which causes redundant data and consistency problems. Adapters are a third option. For example D2RQ[3] provides a relational database as an RDF graph. However, we chose to use our custom representation, as relationship evaluation causes less CPU utilization and sharing relationship functions across processing nodes is easier. Reusing existing ontologies/domain models is still possible, by simply registering them as an external data source.

Data Source Registry The data source registry keeps track of all data sources. The example from Section 1 illustrates that the suggested middleware has to integrate various data sources. We distinguish between two categories of data sources: pull sources (e.g., databases) and push sources (e.g., sensors). Pull sources follow a request/reply paradigm, while push sources typically interact in a publish/subscribe fashion.

Data sources need to be annotated with metadata. The metadata specifies about which concept the data source provides information and which format the data have. For example, the employee table provides information about the concept person.

The position of sensors is useful information and thus also included in the metadata. Unlike location-based publish/subscribe [12, 23], we use location information for enrichment, not routing.

Depending on the kind of data source, we distinguish between *static* and *dynamic* attributes: pull sources provide comparatively static, queryable information, while push sources provide volatile, high frequency data in an event-based fashion. Thus, we call attributes with associated pull sources *static* and those with associated push sources *dynamic* attributes.

Policy Manager The policy manager keeps track of all registered policies. It is responsible for handling new and edited policies, as well as ensuring proper shutdown of all related components when a policy is deleted.

When a policy is created or edited, the policy manager creates a *policy logic tree*, based on the policy and the domain model. The policy logic tree represents

[3] http://d2rq.org/

the policy's concepts, their attribute constraints and relationships with other concepts. The tree then enables the policy engine to make educated judgments which data source to use for enrichment and which relationship functions to evaluate.

We want to illustrate this with the running example: Figure 5 shows the policy logic tree for our running example. The policy logic tree is then further

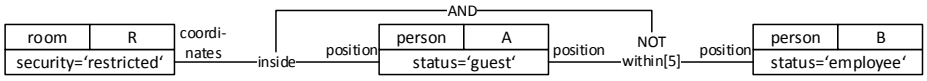

Fig. 5. Policy logic tree for the running example

analyzed. For edges connecting a static attribute with a dynamic attribute (e.g., inside), the policy manager generates an EEC which is triggered for events providing information about the dynamic attribute (e.g., position) and enriches them. For edges connecting two dynamic attributes, the policy manager generates an EEC, which reacts to events from event streams and correlates them. The resulting events are further processed by nodes representing the logic operators. Changing the domain model might force re-analyzing policies and re-generating EECs, which our middleware can handle transparently.

ACTrESS Cyber-physical systems are inherently heterogeneous systems, due to their many different sensors. One example for heterogeneity are different unit systems, e.g., Metric units vs. American Standard units. Thus, an event saying `temperature = 25` can only be interpreted correctly, if producer, broker, and consumer share the same understanding of what `25` means (and what `temperature` means). Heterogeneity becomes even more complicated when we take the structure and types of events into consideration.

In order to correctly interpret events, this heterogeneity has to be mediated. The correct interpretation is vital both for the subscription matching in the broker and handling of events at the consumers. To achieve this, we designed and built ACTrESS (Automatic Context Transformations in Event-based Software Systems), which transforms incoming events to the desired format and interpretation [15]. We proved ACTrESS to be type safe to avoid hard-to-trace runtime errors [16].

ACTrESS enhances message-oriented middleware, by allowing producers and consumers to tell the broker about their interpretation of data (independent of each other). Based on this information, the broker then automatically transforms messages.

For the DPL-enabled middleware we make use of ACTrESS. As pointed out in Section 2.3 producers provide their data format in the metadata. We feed this information into ACTrESS so it can transform incoming events to the needs of the EECs. For example, the generated EECs will always receive absolute

Cartesian coordinates (x, y) , even if a sensor (e.g., a camera) sends events in relative, polar coordinates (r, θ).

EECs We distinguish between query-EECs and correlation-EECs. We illustrate the need for this distinction with our running example:

Checking the `inside` edge of the policy logic tree (c.f. Figure 5) involves knowing a person's and the rooms' coordinates. Since `person.position` is a dynamic attribute (c.f. above), computation is triggered by position events. All other required information (the constraints and the rooms' coordinates) are static attributes and available via pull sources. Upon receiving a position event, the (query-)ECC can query the external sources for the required information and generate an enriched event based on the obtained information.

This is different from the `within` edge. In this case, both sides are dynamic attributes and thus, their information is not readily available when the EEC receives **a** position event. In this case, we must correlate multiple events. Such correlation is best left to event stream processing engines like Esper[4].

We designed EECs so they do not need to run on the same machine as the middleware. This helps in distributing processing and keeping the overall system scalable. EECs therefore subscribe to events from the message bus. At least one EEC per policy is also responsible for invoking the specified action when all conditions are met. Since EECs fulfill a certain event processing related task (enriching events/ taking action), they can be implemented with Eventlets[1]. Eventlets encapsulate tasks in event-based systems and have a managed lifecycle. This allows for instantiating one EEC per instance (e.g., one for every person), resulting in an automatically managed, highly modular (and thus parallelizable and scalable), distributed infrastructure.

The policy engine keeps track of all running EECs. We can stop an EEC if its corresponding policy was deleted. The policy engine also allows for getting various statistics about the EECs, e.g., how often they triggered the policy or even the triggering frequency. We use publish/subscribe monitoring tools like ASIA [17], to keep the monitoring overhead to a minimum.

2.4 Detailed Walkthrough

In this section, we want to illustrate the interaction between the components with our running example from Section 1. We assume that the system has been setup by an expert (e.g., loading the domain model and annotating data sources).

After receiving the policy, the policy engine analyzes it and generates three EECs, based on the policy logic tree: (1) A query-EEC, subscribing to position events and checking (by querying external sources) if the reported position belongs to an employee. If successful, the EEC generates an `EmployeePosition` event. (2) A query-EEC, subscribing to position events and checking if the reported position belongs to a guest and if that guest is inside a restricted

[4] http://esper.codehaus.org/

room. Upon successful detection, the EEC generates a `GuestInsideRestricted` event. (3) A correlation-EEC, subscribing to `EmployeePosition` events and `GuestInsideRestricted` events and checking (using event stream processing) if the received events show a pattern which satisfies the policy's conditions.

Position sensors (e.g., RFID readers) produce `PositionEvent`s as shown in Figure 6. As discussed in Section 2.3, we use ACTrESS to transform the local position to a global one. After this transformation, the broker delivers the position event to the two query-EECs. Depending on the status of the person with id 5 (obtained by querying an external source), one of them generates an enriched event (c.f. Figure 6).

```
<PositionEvent>
  id = 5
  at = (5, 3)
</PositionEvent>
```

```
<EmployeePosition>
  id = 5
  at = (27.4, 40.3)
</EmployeePosition>
```

```
<GuestInsideRestricted>
  id = 5
  at = (27.4, 40.3)
  room = 205
</GuestInsideRestricted>
```

Fig. 6. Source event (left) and enriched events

For guests, the EEC queries an external source to obtain the room which contains the position $(27.4, 40.3)$. Finally, the correlation-EEC uses a stream processing engine like Esper to correlate both event streams (e.g., with the EQL statement given in Figure 7)

```
select 1 from pattern
  [a=GuestInsideRestricted and not b=EmployeePosition]
where within(a.at, b.at, 5)
```

Fig. 7. EQL statement (simplified) used by the EEC

When the EEC detects such a pattern, it generates a `PolicyAction` event, which the middleware detects and invokes the alarm actuator.

3 Case Study

To strengthen and support our initial claim of simplifying the development of cyber-physical systems and event-based systems in general, we use a case study to compare three approaches along several criteria. The three competing approaches are Java, EQL (Event Query Language used by Esper) and DPL. They represent a solution to the evaluation example we give in the next section.

We do not provide a detailed performance analysis due to space reasons. In light of modern cloud infrastructures, single node performance is less important. Thus, we show that our approach is fully distributable (and thus parallelizable).

The generated EECs are autonomous components, which run independently of each other. As indicated in Section 2.3, EECs can be implemented as Eventlets. The execution of each EEC is thereby even further distributed. For example, the Eventlet middleware separates the guest-EEC along `id`. Each EEC instance then processes events for only a single person. The Eventlet middleware automatically manages distribution of instances efficiently. We thus achieve a very fine grained distribution/parallelism, which makes our approach easily scalable with virtually any load.

3.1 Complete Example

For the evaluation, we slightly expand the example from Section 1. There are publicly accessible areas in the company. For all other rooms, guests must be accompanied by employees. Furthermore, there exist restricted areas where guests are not allowed under any circumstances. We can express this in **DPL**:

```
IF
  person A with attribute status='guest' IS INSIDE
  room with attribute security != 'public' AND
  person B with attribute status='employee' IS NOT WITHIN 5m of A   OR
    A IS INSIDE room with attribute security = 'restricted'
THEN
  sound alarm
```

The exact **EQL** statements depend on the specifics of the stream processing engine, the host language and how much functionality is pushed to the streaming system. Thus, we outline how to achieve the above goal and try pushing as much functionality to the streaming system as possible.

We need to make data from the database available to the streaming system, by inserting them into a stream (for every table we need data from):

```
insert into UserStream select * from pattern[timer:interval(0)],
  sql:db1 ['select * from users']
```

We need to generate higher-order events based on the conditions, for example:

```
insert into NonPublicEvent select p.id, p.position, r.room
  from PersonPositions as p, RoomStream as r
  where inside(p.position, r.coordinates)
    and p.status = 'guest' and r.security != 'public'
```

The function `inside` is a host language function made accessible to the streaming system. This EQL statement assumes that there is a stream PersonPositions which combines position events with user data. This illustrates the dependency between different EQL statements. For our example, various such higher-order events are necessary, whose definition we omit for brevity.

Finally, an alarm triggering mechanism based on the given conditions must be specified:

```
insert into AlarmEvents select * from pattern
  [a=NonPublicEvent and not b=EmployeeEvent
   or c=ForbiddenEvent]
where within(a.position, b.position, 5)
```

The host language can subscribe to events matching this pattern, and upon reception of an event, trigger the alarm.

We omit the **Java**-only code for implementing the given scenario, as we believe the reader can imagine the effort to implement this in Java without any stream processing support (like Esper provides).

3.2 Criteria

We want to evaluate our three candidate solutions with the criteria **description similarity**, **number of instructions**, **control**, and **change**. We explain each criterion in more detail:

Description similarity indicates how similar the solution is to the original task (written in prose). We used the MCS method [26] which measures text similarity on a semantic level. Similarity is measured on a scale from 0 to 1, with a higher score meaning a higher similarity. We believe this metric to be a good indicator of how close a solution is to the mental model of the user.

Number of Instructions refers to the *length* of the solution. We use lines of code the solution requires as our metric. We do not count boilerplate code or setup-related code (e.g., registering JMS clients).

Control means how much direct control the user has over what happens inside the system. For example, choice of index structures and data structures.

Change indicates how well changes are supported. Changes might occur because the user wishes to change the behavior, but may also result from changes to the system (e.g., the addition of new sensors). We assume the presence of other policies in the system, which also need to work.

3.3 Results

Table 1 summarizes our results, which we detail in this section.

Table 1. Results of the Case Study

	Java	EQL	DPL
Description similarity	0.08	0.33	0.62
Number of Instructions	158	34	9
Control	++	+	-
Change	--	-	++

Looking at the description similarity, the advantages for DPL are apparent. While it is nowhere close to a natural language definition, DPL still matches the description in some terms and the line of thought. EQL requires the definition of more complex events and access to database data, all of which is not part of scenario description. Java's syntax prevents coming close to a description in prose.

Implementing the given scenario with EQL requires considerably more instructions than using DPL. Many of the EQL statements are simply necessary to prepare the raw data so it can finally be used in a pattern-statement. By using DPL, the middleware takes care of all preparation through enrichment and lets the user focus on the important part. Thus, DPL requires much less effort to implement the scenario. Since the Java implementation cannot rely on stream processing libraries, it takes even more instructions.

On the other hand, more middleware-enabled functionality means less control for the end user. Thus, Java provides the most control, while stream processing libraries usually provide many optional settings. DPL clearly provides little control about how a policy is executed. Providing mechanisms for more control, would complicate the language and execution, which is not our goal.

Changing policies is also much better supported in DPL. For example, if we want to extend the restrictions to interns, in EQL, we will have to change the definition of a `NonPublicEvent`. However, other parts of the system might still depend on the current definition. Thus, the user has to either introduce a new definition besides `NonPublicEvent` with the risk of doing unnecessary work (if `NonPublicEvent` is in fact not used anywhere else), or the user has to analyze all other EQL-statements to see if and how they depend on `NonPublicEvent`. In Java, the user must analyze the code to check where the modifications need to be made and which other parts of the code might be affected, requiring similar effort. Assuming a correct setup of the additional knowledge (e.g., domain model), with DPL, the user simply modifies the query and leaves the rest to the middleware.

In summary, we see the arguments above as good and convincing evidence that DPL indeed, simplifies the development of event-driven architectures.

4 Related work

Complex Event Processing (CEP) [24] is an active field of research. It is based on stream processing and active databases, aiming at providing higher-level abstractions with event patterns, filtering and aggregation. Low-level events are combined according to rules. There is a plethora of complex event processing and stream processing systems [9]. All systems however, try to improve performance [29] or the expressive power of their rule language. Other works are on improving the rule language design itself [22], but stick with the same concept. Eckert et al. surveyed CEP languages [11], but all of them operate on a lower abstraction than our approach and are thus comparable with EQL. Some approaches advocate integrating event processing directly into major languages like Java [13]. Reactive Programming [3] extends this idea by providing more built-in language support to avoid common problems such as value propagation inconsistencies. However, none of these approaches attempts changes as radical as our approach.

Situation monitoring suggests such a perspective shift [4, 18]. They argue that users typically think of situations a computer system should be aware of and then define reactions based on them. They provide tools for defining these situations.

Their approach faces some drawbacks in light of today's distributed, dynamic and heterogeneous systems: their approach does not use message-oriented middleware for forwarding events and they hard-wire information flows from specific sensors to specific processing components. Similarly, behavioral programming aims at "constructing reactive systems [...] from their expected behaviors" [20]. They advocate thinking about a system as the composition of its behaviors, where behaviors are reactions to events. However, they rely on explicitly defining the interactions and an a-priori knowledge of which events exist. Schiefer et al. developed a graphical tool for specifying event-condition action rules [28]. While their approach certainly helps tackling the complexity of event-based systems, it still requires the technical knowledge of how events can be combined.

The Alarm Correlation Engine (ACE) allows for declaratively specifying conditions and actions for alarms in a computer network, based on events [30]. Their motivation, too, is to give domain experts a tool for specifying policies. However, their approach is highly tailored to networks, with a predefined correlation database. Their solution is too rigid for modern event-driven architectures. Handling multiple, heterogeneous data sources with the help of an ontology has been explored in the field of query formulation [25, 8]. Users are supported with suggestions to the queries they are trying to construct. Among other hints, query formulation systems suggest vocabulary based on the underlying ontology.

Higher-level approaches for simplifying the development of cyber-physical systems are Ukuflow, Event Stream Processing Units and MobileFog. Ukuflow allows users to define workflows, which are then deployed and executed in a wireless sensor network (WSN) [19]. Although targeted to WSNs, we believe the approach can be adopted to event-driven architectures in general. Similarly, Event Stream Processing Units (SPUs) support developers and domain experts by introducing an abstraction for event stream processing. Thus, event streams can easily be integrated into the business process modeling. MobileFog abstracts from the distributed nature of applications for event-driven architectures (called *Future Internet Applications*) [21], based on the paradigm of *fog computing* [5]. All three approaches rely on a procedural approach, while we use a declarative approach, each with its unique advantages.

Policies can be viewed as a grouping mechanism of CEP queries. For example, the policy given in Section 3.1 can be seen as a collection (group) of the EQL statements (not technically, but conceptually). In that regard, our approach is similar to constraint grouping techniques in DBMSs [7, 6]. Constraint grouping aims at grouping database constraints into meaningful units, which can be plugged in and out, without worrying about other dependencies. However constraint grouping and query formulation have been designed for pull-based interactions, while we specifically target push-based interactions.

5 Conclusion

In modern software architectures SOAs and Event-driven architectures coexist. While the former has good architectural support the latter still suffers from a

lack of good abstractions. Particular challenges are distribution and no direct control over the control flow. In this work we proposed a novel approach for handling this complexity. Our approach allows users to focus on the desired behavior of the event-driven parts, leaving architectural concerns like distribution to the middleware. We allow for stating the behavior in a declarative way, expressed in DPL. We implemented a prototype of our approach on top of a message-oriented middleware used as a communication infrastructure. We rely on computer experts to do the initial setup of the system, similar to a database developed by database experts for use by domain experts. Results from our case study show, that our approach makes event-driven architectures much easier to use.

Our approach yields some tradeoffs: using a declarative approach allows for easy specification but also means less control for the user. For example, window sizes or the degree/strategy of distribution is beyond the user's control. In terms of system qualities, our approach clearly favors scalability over some others: by automatically dividing the processing into small, independent processors, scaling out is handled automatically, which is especially useful for cloud environments. On the other hand, since we access messages and create new ones, encryption and signatures require trust in our middleware. Even with trust, the necessary key/certificate management becomes a challenge.

In future work, we want to exploit existing spatial database technology and incorporate enhanced event detection from image processing. Furthermore, we want to explore the benefits of dynamically moving EECs to other nodes in the system when CPU load becomes too high. Especially for settings where multiple users interact with a large system, we want to develop a policy conflict detection. This involves making the meaning of actions (and probably also conditions) explicit to the policy engine. We can then use standard conflict resolution strategies to avoid inconsistent system behavior.

Acknowledgment. Funded in part by the German Federal Ministry of Education and Research (BMBF) grant 01IS12054. The authors assume responsibility for the content.

References

1. S. Appel, S. Frischbier, T. Freudenreich, and A. Buchmann. Eventlets: Components for the Integration of Event Streams with SOA. In *SOCA*, December 2012.
2. S. Appel, S. Frischbier, T. Freudenreich, and A. Buchmann. Event stream processing units in business processes. In F. Daniel, J. Wang, and B. Weber, editors, *BPM*, volume 8094 of *LNCS*, pages 187–202. Springer Berlin Heidelberg, 2013.
3. E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45:52:1–52:34, 2013.
4. D. Baker, D. Georgakopoulos, M. Nodine, and A. Cichocki. From events to awareness. In *Services Computing Workshops, 2006. SCW'06. IEEE*, pages 21–30, 2006.
5. F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *MCC*, 2012.

6. A. Buchmann, R. S. Carrera, and M. A. Vazquez-Galindo. A generalized constraint and exception handler for an object-oriented CAD-DBMS. In *OODS*, 1986.
7. A. Buchmann and C. P. de Célis. An architecture and data model for CAD databases. In *VLDB*, 1985.
8. T. Catarci, T. Mascio, E. Franconi, G. Santucci, and S. Tessaris. An ontology based visual tool for query formulation support. In *OTM Workshops*. Springer Berlin Heidelberg, 2003.
9. G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.
10. U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented database system. In K. Dittrich, editor, *AOODS*, volume 334 of *LNCS*, pages 129–143. Springer Berlin Heidelberg, 1988.
11. M. Eckert, F. Bry, S. Brodt, O. Poppe, and S. Hausmann. A cep babelfish: Languages for complex event processing and querying surveyed. In S. Helmer, A. Poulovassilis, and F. Xhafa, editors, *Reasoning in Event-Based Distributed Systems*, volume 347 of *SCI*, pages 47–70. Springer Berlin Heidelberg, 2011.
12. P. Eugster, B. Garbinato, and A. Holzer. Location-based Publish/Subscribe. In *4th IEEE International Symposium on Network Computing and Applications*, 2005.
13. P. Eugster and K. Jayaram. Eventjava: An extension of java for event correlation. In S. Drossopoulou, editor, *ECOOP*, volume 5653 of *LNCS*, pages 570–594. Springer Berlin Heidelberg, 2009.
14. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.
15. T. Freudenreich, S. Appel, S. Frischbier, and A. Buchmann. ACTrESS - automatic context transformation in event-based software systems. In *DEBS*, 2012.
16. T. Freudenreich, P. Eugster, S. Frischbier, S. Appel, and A. Buchmann. Implementing federated object systems. In G. Castagna, editor, *ECOOP*, volume 7920 of *LNCS*, pages 230–254. Springer Berlin Heidelberg, 2013.
17. S. Frischbier, A. Margara, T. Freudenreich, P. Eugster, D. Eyers, and P. Pietzuch. Aggregation for implicit invocations. In *AOSD*, 2013.
18. D. Georgakopoulos, D. Baker, M. Nodine, and A. Cichoki. Event-driven video awareness providing physical security. *World Wide Web*, 10(1):85–109, 2007.
19. P. Guerrero, D. Jacobi, and A. Buchmann. Workflow support for wireless sensor and actor networks. In *DMSN*, 2007.
20. D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Commun. ACM*, 55(7):90–100, July 2012.
21. K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe. Mobile fog: a programming model for large-scale applications on the internet of things. In *MCC*, 2013.
22. T.-G. Le, O. Hermant, M. Manceny, R. Pawlak, and R. Rioboo. Unify event-based and rule-based styles for developing concurrent and context-aware reactive applications. Technical report, LISITE, France, 2012.
23. G. Li, Y. Wang, T. Wang, and J. Feng. Location-aware publish/subscribe. In *SIGKDD*, 2013.
24. D. C. Luckham and B. Frasca. Complex event processing in distributed systems. Technical report, Stanford University, 1998.
25. K. Mahalingam and M. Huhns. An ontology tool for query formulation in an agent-based context. In *COOPIS*, 1997.
26. R. Mihalcea, C. Corley, and C. Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *AAAI*, volume 6, pages 775–780, 2006.

27. G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*, volume 1. Springer Berlin Heidelberg, 2006.
28. J. Schiefer, S. Rozsnyai, C. Rauscher, and G. Saurer. Event-driven rules for sensing and responding to business situations. In *DEBS*, 2007.
29. E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *ACM SIGMOD*, 2006.
30. P. Wu, R. Bhatnagar, L. Epshtein, M. Bhandaru, and Z. Shi. Alarm correlation engine (ACE). In *NOMS*, 1998.