2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)

Quality-Aware Runtime Adaptation in Complex Event Processing

Pascal Weisenburger*, Manisha Luthra†, Boris Koldehofe†, Guido Salvaneschi*
Technical University of Darmstadt, Germany
*{lastname}@cs.tu-darmstadt.de, †{firstname.lastname}@kom.tu-darmstadt.de

***Abstract*—Complex event processing (CEP) is a fundamental paradigm for a software system to self-adapt to environmental changes. CEP provides efficient means to detect (complex) events corresponding to environmental changes by performing a real-time analysis on many, possibly heterogeneous, data sources. The way current CEP systems detect events is determined at design time without accounting for dynamic changes of the environment monitored by the CEP system. This can lead to situations where the performance, quality and reliability of event detection significantly drop (e.g., due to mobility) since initial assumptions of the environment are violated or stated too general. In this paper, we propose AdaptiveCEP, a CEP system that is able to self-adapt to detected changes in environmental conditions. We propose a CEP query language that allows specifying changes in the behavior of the CEP system and its mechanisms in detecting events dependent on environmental conditions. This way, AdaptiveCEP can select the best-suited configurations for given quality demands. In our evaluation, we show by means of a reference concept how the flexibility exposed by the query language helps to achieve significant performance gains.**

I. Introduction

The behavior and functioning of software systems in many cases depend on the environment in which they operate. Providing methods to adapt dynamically to the environment is, therefore, fundamental to ensure correct and efficient operation. Adaptation of software systems [38], [9] has been achieved in many different ways, e.g., with component-based software architectures, aspect-oriented programming, metaprogramming and agent-oriented languages [26], [42].

A key aspect in adapting a software system is to detect environmental changes to which the software system needs to react. In doing so, software systems deal with incoming streams from an increasing number of data sources such as stationary and mobile devices, as well as maintain state information of the devices and software components. In the age of the Internet of Things and with the significant rise of the mobile devices (predicted to be two-third of the total traffic by 2020 [10]), software systems need to analyze huge and heterogeneous input streams. For most of the applications, this analysis has to be in real-time, to timely detect events of interest corresponding to the changes in the environmental conditions. To efficiently capture events of relevance to a software system, *Complex Event Processing* (CEP) has emerged as the paradigm of choice. CEP supports the detection of so-called *complex events* derived from primary event sources. CEP is a fundamental building block for very large and highly dynamic and distributed software systems and is used in a wide field of applications like finances, traffic monitoring, monitoring data centers and logistics [19].

CEP allows for the specification of *event patterns* occurring in time-changing event streams by means of a query language. For example, to trigger an alarm, an event pattern may comprise a sequence of k outliers within the last n events of the stream or within a timespan of duration T. Event patterns like the alarm may be used for the specification of other event patterns. With CEP, it is therefore relatively easy to specify large software systems through event composition. In addition, CEP also offers ways to efficiently notify software systems about detected events. This is typically achieved with the help of an execution environment in charge of executing event processing operators. Operators select appropriate event streams and detect complex events for specific event patterns. According to the composite nature of an event specification, the operators form an operator graph given by operators (nodes) and event transmission among them (edges).

While occurrences of events are very dynamic in nature and the execution environment may support dynamic mapping of operators to hosts, CEP systems typically do not allow developers to specify requirements on quality attributes depending on the environmental conditions. This imposes major limitations on the ability of the CEP system to self-adapt in a highly dynamic environment, e.g. in a system environment with mobile devices. In such scenarios, the assumed static properties like the type of data sources may change (e.g., due to device mobility), and in turn require changes to the composition of event patterns and the operator graph. For instance, the maximum latency until an alarm is detected depends on the location of the users to be notified. Being aware of environmental changes offers a high potential for self-adapting CEP systems. This is done by dynamically selecting the best-suited mechanisms and algorithms for placement and execution of operators. Thus it is necessary to identify conditions that trigger adaptations. That allows a CEP system to be self-adaptive to new situations and self-optimizing under changing conditions.

In this paper, we aim to enable CEP to self-adapt to the dynamic environmental changes it encounters. We present the design and implementation of AdaptiveCEP, a CEP system that automatically adapts to predicted changes in the environment. This way, the system is capable of ensuring a given level of quality by adapting to a configuration, which can fulfill those demands. The AdaptiveCEP language allows developers to define an adaptive CEP system in a concise, high-

978-1-5386-1550-8/17 $31.00 © 2017 IEEE
DOI 10.1109/SEAMS.2017.10

level way, easing development and maintenance of this class of systems. The language accounts for functional requirements such as event processing computations but – contrarily to existing languages [12] – also for non-functional requirements like quality constraints for event delivery. We design our query language and system utilizing user knowledge on the variability of the environment and depending quality attributes, e.g., latency, to adapt the CEP system at run time.

To the best of our knowledge, none of existing CEP systems is equipped with programmable self-adaptation, nor exists any CEP language that allows developers to specify high-level requirements for the system. Specifically, in this work, we make the following contributions:

- We propose a self-adaptive CEP system, enabling the use of CEP in highly dynamic environments.
- We present a CEP query language embedded into Scala which allows specifying queries over event streams as well as quality demands that the system needs to fulfill.
- We design a runtime environment which allows expert developers to access the CEP system configuration via an interface based on Functional Reactive Programming (FRP) and to specify new adaptation strategies.
- We provide a reference implementation of AdaptiveCEP on top of Akka actors and evaluate our approach showing that with AdaptiveCEP the CEP system is capable of remaining effective in changing conditions.

In this work, we focus on methods providing the ground for self-adaptation in CEP. Since these can be applied to a wide range of adaptation strategies it is out of scope to investigate specific (advanced) adaptation strategies (e.g, [7]) or the combination of multiple strategies [16] – both subject of extensive research in the past. Plugging those results into our framework to achieve better adaptation results is planned for future work.

The paper is structured as follows: Section II provides background on CEP and FRP. Section III gives an overview of the AdaptiveCEP system and its query language. Section IV describes the API available to expert developers. Section V provides insights into the implementation. Section VI discusses the evaluation. Section VII presents related work. Section VIII concludes and outlines future work.

II. Background Section

A. Complex Event Processing

A CEP system comprises both (i) methods to specify complex events in form of a query language and (ii) methods to perform the efficient detection of events. Consider for example a video game based on augmented reality. In this scenario, players send updates for their current game state, e.g., if they are actively playing or have paused the game. Hence, the `playerInfoStream` carries player state events, which have a `gamePaused` field. A central coordinator keeps the game score for all players. Updates to the player's score are sent over `gameInfoStream`. Players need to be constantly updated about the actions of the other players. A possible query that players may submit to the system is illustrated in Figure 1. The query generates a complex event stream `infoStream` that comprises state information of all active players. The state is composed of local state updates from the players and updates on the score produced by the central coordinator. The resulting complex event stream carries the score for every player that is currently active, i.e., players who have not currently paused their game instance.

```
infoStream :=
  (playerInfoStream WINDOW 30 sec SLIDING)
  JOIN (gameInfoStream WINDOW 30 sec SLIDING)
  ON (playerInfoStream.id == gameInfoStream.playerId)
  WHERE NOT gamePaused
```

Figure 1: The online game query.

A sliding window collects all events of `playerInfoStream` and `gameInfoStream` that occurred within in a period of 30 s. Over the events in the window, a join (⋈) operator (`JOIN` clause) links information from `playerInfoStream` and `gameInfoStream` based on the value of the `id` attribute used in the first stream and the value of the `playerId` attribute used in the second stream to identify the player. The selection (σ) operator (`WHERE` clause) ensures that only state of active players is forwarded by filtering all events where the value of the attribute `gamePaused` is false. The execution environment of the CEP system, therefore, has to execute multiple operators. The corresponding operator graph for the query – imposing the flow of events from producer to consumer – is shown in Figure 2.

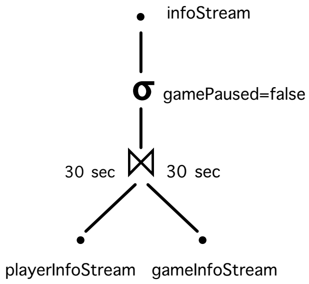


Figure 2: Operator graph for the online game query.

The performance of a CEP system will depend on (i) how the query is specified, (ii) how the query is mapped to the operator graph and (iii) how the operators are mapped to available hosts of the software system. In the system design for AdaptiveCEP, we will mainly aim to improve the adaptivity of CEP by improving the specification of queries. However, as we will show later, this offers also higher flexibility and the potential for the subsequent steps in adapting a CEP system.

B. Functional Reactive Programming

FRP has been introduced to address the issues of the Observer design pattern, which has been criticized in literature for long. The main drawbacks include (i) lack of composability as callbacks return void, (ii) need of globally updated variables and (iii) complexity of analysis and comprehension due to inversion of control [28]. FRP has been introduced in the context of animations [18] and it has been successfully applied to other areas including user interfaces [28], [13], robotics [20] and sensor networks [25].

In FRP, *signals* represent continuous time changing values which are automatically updated by the language runtime. For example, in the following code snippet:

```
val position: Signal[(Int, Int)] = mouse.position
```

the `position` signal always contains the updated value of the mouse position. Expressions that contain signals are also automatically updated by the runtime. For example in the following code, the `shifted` signal always contains a mouse position that results from shifting the real one.

```
val shifted: Signal[(Int, Int)] =
  Signal { mouse.position() + (10, 10) }
```

Signals inside `Signal{...}` expressions require `()` to trigger the DLS machinery that registers them as dependencies. In FRP, discrete time changing values are represented as events. Events and signals can be combined. The following code snippet defines a signal for the mouse position of the last click by snapshotting the mouse position signal when the click event occurred.

```
val clicked: Event[Unit] = mouse.clicked
val lastClick: Signal[(Int, Int)] =
  position snapshot clicked
```

We decided to use FRP for the interface of our system for expert developers (Section IV) in the cases where the system configuration can be conveniently modeled with time-changing values – FRP provides abstractions to compactly process them.

III. AdaptiveCEP Query Language

AdaptiveCEP provides a CEP query language that developers of event-based adaptive systems can immediately use. Expert developers may also want to program the internal details of the adaptation strategies. In this section, we present the AdaptiveCEP system and query language to define complex event streams and their quality demands. The next section presents a more advanced use of AdaptiveCEP introducing more fine-grained details of the adaptation process.

A. A Bird's Eye View of AdaptiveCEP*.*

Considering the augmented reality gaming scenario (cf. Section II-A), additional requirements arise in a real-world setting. For example, the scenario requires low communication latency to keep the information about the other players constantly up to date.

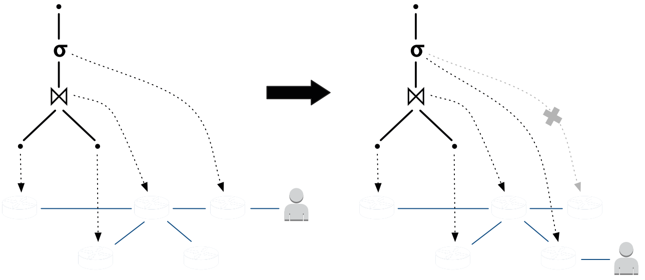
The query in Figure 3 provides the same basic functionality as the query in Figure 1. However, it takes into account that the information of proximate players in the game requires synchronization closer to real-time and therefore updates from these players have stricter latency demands. Events for players in close proximity, i.e., within the range of 100 m (Line 7), should be made available locally with a latency lower than 50 ms (Line 6). The query is submitted to the system using the `run` method (Line 12). Noticeably, quality demands do not change the functional dependencies in the operator graph. As a result, the operator graph for the query in Figure 3 is still the one in Figure 2.

```
1   val playerInfoStream: Stream[PlayerInfo] = ...
2   val gameInfoStream: Stream[GameInfo] = ...
3
4   val infoQuery: Query[ResultInfo] =
5     ((playerInfoStream window 30.sec.sliding)
6          demand (latency lower 50.ms when
7                 (proximity within 100.m))
8       join (gameInfoStream window 30.sec.sliding)
9       on ('id === 'playerId)
10      where { infoEvent => !infoEvent('gamePaused) })
11
12  val infoStream: Stream[ResultInfo] = infoQuery.run
13
14  infoStream fired { result: ResultInfo =>
15    updateGame(result) }
```

Figure 3: The online game query in AdaptiveCEP.

Further details about the query are given in Section III-B. For now, it suffices to say that after being submitted, the query becomes standing and outputs an event stream, which can be directly consumed or composed with other streams in a different query. User code can attach event handlers to the stream produced by a query (Line 14). The handler is executed whenever an event in the stream is fired.

The operators in the operator graph are mapped to hosts (e.g., servers, player's mobile devices, or network elements) along the physical path from the consumers to the producers. As pointed out in Section II-A, this mapping is crucial for the resulting performance. For example, deploying an operator on an unloaded host can increase its processing rate, and selecting event sources on hosts that are physically close can significantly decrease event communication latency.



(a) Initial deployment. (b) Deployment after transition.

Figure 4: Transition between deployment configurations.

A possible deployment for the operator graph of Figure 2 representing the query in Figure 3 is shown in Figure 4a. Dotted arrows indicate on which hosts operators are deployed. Event producers, i.e., the stream from the central game controller and a stream from another player, run on different network devices. Similarly, the join and the filter operator are placed on separate hosts. Finally, the host for the filter operator provides the stream to the end user device.

The event processing system defined by AdaptiveCEP is self-adaptive thanks to the capability of reconfiguring the operator graph and its mapping to the underlying physical infrastructure to fulfill the demands expressed in the query. The

Operator	Subquery
Stream source	*stream*
Join	(*query0* window *win0*) join (*query1* window *win1*) on *join-condition*
Filter	*query* where *predicate-function*
Map	*query* map *mapping-function*
Aggregation Min	*query* min *numeral-selection-function*
Aggregation Max	*query* max *numeral-selection-function*
Aggregation Avg	*query* avg *numeral-selection-function*
Logical And	*query0* && *query1*
Logical Or	*query0* || *query1*
Logical Not	!*query*
Temporal Sequence	*query0* -> *query1*

Table I: CEP Operators.

Demand	Subquery
Demand on stream	*stream* demand *demand*
Latency	latency lower *latency-value*
Throughput	throughput higher *throughput-value*
Bandwidth	bandwidth higher *bandwidth-value*

(a) Quality demands

Condition	Subquery
Condition on quality demands	*demand* when *condition*
Condition on event producers	*stream* only *condition*
Proximity	proximity within *length* proximity nearest *count*
Frequency	frequency higher *frequency-value*

(b) Quality conditions

Table II: QoS Operators.

reconfiguration is based on a number of constraints, including global performance optimizations, e.g., event selection moved close to the sources, and local performance optimizations like avoidance of high-latency communication paths.

The adaptation of the operator graph spans over different axes. Operators that are shared by different queries can be factorized and merged into single operators that serve both queries, avoiding duplicated processing of the same events. On the other hand, operators can also be duplicated to increase processing speed. Operators can be migrated from one host to another to satisfy quality demands, e.g., end-to-end latency – possibly at the cost of other queries with more relaxed end-to-end latency demands. We greatly take advantage of approaches for operator migration discussed in detail in literature [33], [35], [36]. In case of leaving nodes, we may encounter a quality decay until the system has adapted itself.

In our example query (cf. Figure 3), communication latency between players depends on the distance between them. A possible adaptation for the query is depicted in Figure 4. Compared to the initial operator placement (cf. Figure 4a), the end user moved to another position (cf. Figure 4b). This increases latency for the connection to the host on which the filter operator is placed. To keep the latency demand satisfied, the filter operator is migrated to a host closer to the new location of the user.

B. Query Language

As we have seen, in AdaptiveCEP, programmers express processing of event streams via continuously active queries. Each query defines event processing operators that receive event streams and produce new streams of complex events.

The fundamental abstraction in AdaptiveCEP is a `Stream`, which is used to model event flows. The AdaptiveCEP query language provides operators on streams to compose them, correlate events and process their payload. Event streams are statically typed. The event stream `playerInfoStream` (cf. Figure 3) carries events of type `PlayerInfo` (Line 1) and the event stream `gameInfoStream` carries events of type `GameInfo` (Line 2). Every event is a timestamped record of possibly multiple fields with possibly different types. The fields of the record can be named, e.g., the example query accesses the `id` field of `playerInfoStream` (Line 9) and the `playerId` and `gamePaused` fields of `gameInfoStream` (Line 9 and 10). Running the query produces the resulting stream `infoStream` (Line 12). Besides creating event streams through composition, events can be fired explicitly using the `Stream`'s `fire` method, which can be used to create event sources. For example, players can fire events on their `playerInfoStream` to provide updates to other players by using `playerInfoStream.fire(myInfo)`. The system supports CEP operators, which define the processing logic, but also QoS operators to specify quality demands.

1) CEP Operators: AdaptiveCEP supports (i) operators typically found in the event algebras of CEP systems, i.e., disjunction and conjunction of events, sequencing and windowing to recognize event patterns, (ii) aggregations, i.e., minimum, maximum and average, and (iii) relational operators, i.e., selection, projection and joins of event streams. Table I lists the available operators and shows how they are applied to streams. Queries can be composed using the listed operators, which have one or more subqueries as children, forming a query tree with event streams in its leaves. The example query composes two streams `playerInfoStream` (Line 5) and `gameInfoStream` (Line 8) by joining them (Line 8) and selecting only events which satisfy a given predicate (Line 10).

2) QoS Operators: In addition to CEP operators, AdaptiveCEP also provides QoS operators that programmers can adopt to specify non-functional requirements for the system, e.g., to declare a minimum frequency for emitting events, or latency bounds for delivering events to the consumer. Table II presents the supported QoS operators. Quality demands can be specified for streams as given in Table IIa, e.g. `playerInfoStream demand (latency lower 50.ms)`. The first row in the table shows how to specify any quality demand for a stream using the `demand` clause. The next rows show the available demands.

Quality demands can be *conditional* as shown in Figure IIb, i.e., they can only apply if a given condition is true, e.g., `playerInfoStream demand (latency lower 50.ms) when (proximity within 100.m)`. The first row in the table shows how to specify a condition for a quality demand using

```
trait Demands {
  def violatedDemands: Signal[Set[QoS]]
  def adapting: Signal[Option[Set[QoS]]]
  def adaptationPlanned: Event[Set[QoS]]
  def delayAdaptation(delay: Event[Duration]): Unit
}
```

Figure 5: Demands interface.

the `when` clause and the second row shows how to specify a condition for a stream using the `only` clause. Conditions on streams are used to narrow down the set of producers for the stream whose events are used, e.g., only producers in a certain radius (*stream* `only (proximity within 100.m)`) or only the nearest producer (*stream* `only (proximity nearest 1)`).

The next rows present the available conditions, which are based on information monitored by AdaptiveCEP, e.g., geographic position of hosts and frequency of emitting events. The `proximity` condition restricts the consideration of quality demands or the set of producers based on the spatial proximity between event consumer and producer. The `frequency` condition represents a restriction based on the frequency of events provided by a stream. Conditions are currently not based on complex events themselves.

The example in Figure 3 specifies a quality demand using the `latency` QoS operator in the `demand` clause (Line 6). The demand is conditional, i.e., the latency demand is only taken into consideration based on the `proximity` quality condition (Line 7). A latency of less than 50 ms is only demanded for `playerInfoStream` producers within the range of 100 m of the `infoQuery` consumer.

C. Demands Violation

AdaptiveCEP also provides mechanisms to (i) inform user code about quality violations and (ii) introduce countermeasures to restore the quality of service. When describing the advanced AdaptiveCEP development we will detail the countermeasures. For now, we focus on user code notification. A notification is triggered whenever a quality demand is not satisfied and whenever the system is about to initiate a potentially time-consuming adaptation. This allows the developer to intervene appropriately, e.g., delay the adaptation or notify the end user. Figure 5 presents the `Demands` interface[1]. User code can use this interface to get access to the system's current state, i.e., violated quality demands and current adaptations.

The interface provides the set of currently violated demands as time-varying signal `violatedDemands`. When the system is adapting, the `adapting` signal contains the set of quality demands to satisfy with the adaptation. When no adaptation is in progress, the signal contains an empty `Option`. Whenever an adaptation is planned but not yet performed, the `adaptationPlanned` event is fired. It carries the quality demands to fulfill in the adaptation. User code can choose to delay a planned adaptation by firing an event passed to `delayAdaptation`. The event carries the timespan for which the adaptation should be delayed.

For example, the following code uses the `infoQuery` query from Figure 3 and notifies the player of the game when inconsistencies between his view and the other players' views on the game are anticipated due to high update latencies between players.

```
1 ui.gameInconcistencyWarning = Signal {
2   if (infoQuery.violatedDemands().isEmpty)
3     None
4   else if (infoQuery.adapting().nonEmpty)
5     Some(optimizingMessage)
6   else
7     Some(warningMessage)
8 }
```

The code sets the `gameInconcistencyWarning` signal of the user interface (Line 1). As long as no quality demands are violated (Line 2), no warning is given (Line 3). If demands are currently violated and the system is adapting (Line 4), the player is informed that the game is currently being optimized (Line 5). Otherwise, when quality demands are violated and no adaptation is taking place, e.g., because the system is unable to satisfy the quality demands, the player is informed about potential inconsistencies in his view on the game (Line 7).

D. Language Integration

To increase the usability of the AdaptiveCEP, its query language is designed as a domain-specific language embedded into Scala. The language is independent of the runtime backend as explained in Section IV.

In the AdaptiveCEP query language, queries are first-class values, i.e., they can be used as function argument and return value and be assigned to variables. This way, subqueries can be bound to variables, which can then be used to compose the final query. These variables can just be used in the query without the need of splicing them into a string, which allows the Scala compiler to ensure type-safety. For example, `latency lower` *d* expects a value of type `Duration` for *d* and `proximity within` *l* expects a value of type `Length` for *l*. A query that does not satisfy the type constraints is rejected by the compiler.

A query expression can directly use the abstractions of the host language. For instance, filtering an event stream can be performed using a Scala function as predicate. The example in Figure 3 uses a Scala lambda to filter events (Line 10). Integration with the Scala host language allows the developer to outsource a possibly more complex filter condition into its own function, e.g., `myCondition`, and just pass this function as filter in the query, i.e., `query where myCondition`.

IV. Advanced AdaptiveCEP Development

Expert developers can go beyond the high-level query language presented in Section III in order to design different approaches for handling unsatisfied demands. To control the adaptation of the system at such lower level, we further offer a more fine-grained API. In this section, we describe AdaptiveCEP's interfaces to inspect the state of the system, plug in new strategies and actuate changes to the system based on the applied strategy.

[1] We use the term *interface* to generically refer to an abstract type that defines method signatures. Scala uses the `trait` language construct to model (among other things) interfaces.

```
1  trait CEPSystem {
2    val hosts: Signal[Set[Host]]
3    val operators: Signal[Set[Operator]]
4  }
5  trait QoSSystem {
6    val qos: Signal[Set[QoS]]
7    val demandViolated: Event[QoS]
8  }
9  trait System extends CEPSystem with QoSSystem
10
11 trait Host {
12   val position: Coordinate
13   val neighbors: Set[Host]
14 }
15 trait Operator {
16   val host: Host
17   val inputs: Set[Operator]
18   val outputs: Set[Operator]
19 }
```

Figure 6: System interface.

A. *Reactive System Inspection Interface*

ADAPTIVECEP provides an interface to access the current configuration of the system and the current quality measurements (Figure 6). The `System` interface is composed of the `CEPSystem` interface to access the current hosts and operators and the `QoSSystem` interface to access quality measurements as time-varying signals. Sensor values are modeled as FRP signals since, at any point in time, they represent the system's current state and this knowledge can change over time. Results based on these signals are automatically recomputed whenever a signal changes its value. Quality violations are discrete time occurrences, hence they are provided as events in the `QoSSystem` interface. The `Operator` interface provides access to the upstream and downstream operators and to the host, on which the operator is currently placed. Operators without inputs are event sources and operators without outputs are event sinks. The `Host` interface offers access to the geographic coordinates of a host (if known) and of neighbor hosts, i.e., the next hops.

For example, the following definition of `freeHosts` uses the `System` interface to find all hosts in the system on which no operator is currently placed.

```
val freeHosts = Signal {
  system.hosts() -- (system.operators() map { _.host })
}
```

The code snippet first maps all deployed operators to the hosts on which they are placed via `map { _.host }`. Then it removes those hosts from the set of all hosts currently connected to the system. The resulting signal is automatically re-evaluated whenever the hosts or the deployed operators change.

As an additional example, we show how to compute the (time-varying) mapping from each operator to its potential neighbor hosts and the current latency to each respective host:

```
1 val latenciesToOperatorNeighbors = Signal {
2   (system.operators() map { operator =>
3     operator -> (system.qos() collect {
4       case Latency(operator.host, host, latency) =>
5         host -> latency })
6   }).toMap
7 }
```

For each operator (Line 2), the code collects all latency measurements and the respective host (Line 3). The line `host -> latency` creates a pair, which assigns the measured latency to the host. Finally, each operator is assigned the set of the measured host–latency pairs (Line 3) and these assignments are converted into a `Map` using `toMap`.

B. *QoS-guided Adaptation Strategies*

In ADAPTIVECEP, a *strategy* specifies the adaptation policy to keep quality demands fulfilled based on the usual MAPE-K control loop [22]. ADAPTIVECEP constantly monitors the system's state, i.e., joining and leaving hosts, their positions and current quality measurements according to several parameters such as latency, and update frequency. Quality measurements are analyzed and matched against the specified quality demands. Quality measurements and violations of quality demands are provided to the strategy to plan the system's adaptation. In this paper, we consider adaptation strategies that migrate operators to different hosts. Table III shows the supported demands and exemplary strategies with countermeasures taken by the strategy when the corresponding quality demand is not satisfied.

In the case of latency, a strategy can collect and analyze network topology information to find the shortest or fastest network path and place the operators of a query on hosts such that the query respects the latency demand. In the case of multi-objective optimization, strategies encapsulate the decision-making process of potential trade-offs among multiple demands.

Strategy Development Interface: We adopt FRP for developing strategies because they naturally exhibit a reactive behavior, i.e., they produce adaptations based on the current system configuration (hosts and placement of operators) and current time-changing quality measurements. In the API, a strategy is a function which is given the `System` interface as input and produces an event stream of adaptations. Strategies can also monitor changes in the `qos` signal of the `System` interface (Figure 6, Line 6) to proactively adapt the system when quality measurements indicate that a demand violation is foreseen.

An example for a latency strategy which migrates operators based on the fastest path to the sources is shown in Figure 7. The implementation collects all events for violated latency demands (Line 2), all current latency measurements (Line 7) and uses both information to find possible operator migrations (Line 12) to satisfy the demand. We assume the fastest path is calculated by `placeOnFastestPath`, which returns the necessary actuations, i.e., operator migrations.

The `latencies` signal (Line 7) wraps its computation into a `Signal` expression. The expression accesses the `system.qos` signal (Line 8). Hence, the value of `latencies` depends on the value of `system.qos`. By using reactive abstractions, the value of `latencies` is automatically updated to reflect changes of `system.qos`. For the definition of `adaptation` (Line 12), we use an `Event` expression, which generates a new event for every occurrence of the `latencyViolatedOperator` event (Line 13) and uses the current values of `system.hosts` and `latencies` (Line 14) to compute the new fastest path.

Quality	Strategy	Countermeasure for Violated Demand
Frequency	a) Adapt source frequency b) Choose another source	Notify event source to change frequency for emitting events Find new sources that can satisfy the demand
Latency	a) Choose shortest path b) Choose fastest path c) Choose another source	Find path with minimal number hops Find path with minimal round trip time Find an equivalent source with lower latency
Proximity	a) Choose another source	Find an equivalent nearer source
Throughput	a) Choose path with higher throughput b) Choose multiple intermediate hosts	Find path over hosts which can provide a higher throughput regarding their processing power Find a set of hosts which can provide a higher throughput regarding their processing power when each host processes a slice of all events
Bandwidth	a) Choose path with higher bandwidth	Find path over hosts which can provide whose network link all provide higher bandwidth

Table III: Demands operators in AdaptiveCEP and adaptation Strategies.

```
system: System => {
  val latencyViolatedOperator: Event[Host] =
    system.demandViolated collect {
      case Violation(operator, Latency(_, _, _)) =>
        operator }

  val latencies: Signal[Latency] = Signal {
    system.qos() filter {
      case latency @ Latency(_, _, _) =>
        latency }
  }
  val adaptation: Event[Adaptation] = Event {
    latencyViolatedOperator() map { placeOnFastestPath(
      _, system.hosts(), latencies())) }
  }
  adaptation }
```

Figure 7: Latency fastest-path strategy.

C. QoS Monitoring Service Interface

Different *monitoring services* can be implemented for AdaptiveCEP, which are responsible for measuring the quality of service in the system. Multiple monitoring services for different quality types are simultaneously active. Since accessing the quality values depends on low-level implementation details (e.g., measuring latency requires to interact with the communication system used by the middleware), expert developers are not expected to add new quality measures. Our backend implementation already provides monitoring services for the quality demands supported by AdaptiveCEP. Expert developers can combine the primitives offered by our system to provide more elaborated implementations for quality measurements. More complex implementations could also run AdaptiveCEP queries to monitor the quality of service.

An example for a latency monitoring service, which periodically performs latency measurements is in Figure 8. A monitoring service is a function, which is given the `CEPSystem` interface (cf. Figure 6) as input and produces an event stream of quality measurements. Based on a timeout event `timerEvent` (Line 3), the exemplary monitoring service triggers latency measurements between every host and each of its neighbor hosts. It constructs a new `Latency` value to represent the latency between two hosts. The low-level implementation of `measureLatency` (Line 8) directly interacts with the system backend. A possible implementation could ping each neighbor and base the latency measurement on the ping round-trip time.

```
1 system: CEPSystem => {
2   Event {
3     timerEvent() map { _ =>
4       system.hosts() flatMap { host =>
5         host.neighbors map { neighbor =>
6           Latency(
7             host, neighbor,
8             measureLatency(host, neighbor)) } } } } }
```

Figure 8: Sample implementation for latency monitoring.

V. Implementation

1) Embedded DSL: The AdaptiveCEP query language is implemented based on Scala's DSL support that includes infix notation for methods, implicits and operator overloading [31], which are heavily used to support the syntax of the DSL.

In the language, the payload of each event is represented by statically typed tuples, i.e., events offer index-based access to their fields of different types. Since using a name instead of an index number is more convenient for developers, we allow to assign names to fields by using Shapeless [37], a Scala library which provides advanced typing functionalities. Shapeless supports heterogeneous lists (HLists), which are type-checked based on the specific type of each element. For example, for a generic list of type `String :: Int :: HNil`, the compiler checks that `val a: String = list(0); val b: Int = list(1)` is correct. Since heterogeneous lists in Shapeless are essentially extensible records where a name can be assigned to every element, we use them to model events. The extensibility of the records is particularly useful when joining event streams, where the resulting tuple is a type-safe concatenation of the fields of both joined tuples [23]. The following example joins two streams which carry events of type `(String, Long)` and `(Long, Double)`, respectively:

```
val stream0: Stream[(String, Long)] = ...
val stream1: Stream[(Long, Double)] = ...

val query: Query[(String, Long, Double)] =
   join ((stream0 window 30.seconds.sliding)
         (stream1 window 30.seconds.sliding)
   on (_1 === _0))
```

The streams are joined on the second element of `stream0` and first element of `stream1` using the `_1 === _0` join condition. The compiler checks not only that the types of the fields in the join condition, i.e., `Long` in the example, match, but also that the result type of the query (`String, Long, Double`) is a concatenation of both streams' fields except that the field of the join condition is only present once. This kind of check would not be possible using standard Scala or Java generics. The type of the query can also be inferred by the compiler when not given explicitly.

The developer can directly work with extensible records or plain Scala tuples and use Shapeless' generic programming functionality for converting from/to each other or from/to other isomorphic algebraic data types like Scala case classes.

2) Runtime Environment: We use the Akka actor system [2] to encapsulate and distribute operators. Each actor implements a single operator and streams are implemented via messages passed from one operator to another. Simple operators directly work on the event messages, e.g. filtering is implemented by forwarding only events that fulfill a given predicate. For more complex operators, e.g., joins or sequences, we integrate with the Esper CEP engine [1] to aggregate and correlate events. We do not use another intermediate layer like Akka Streams – which also provide event processing operators – because it lacks support for complex correlations, like joins.

Our implementation of QoS operators is based on the interaction of (i) measuring quality of service via monitoring services for each quality demand and (ii) applying adaptation strategies, which take action depending on the measured values. In the current implementation, the values obtained from monitoring on each host are forwarded to a central coordinator, which executes the strategies that can assume global knowledge of the system's state. For example, our implementation monitors latency by measuring the round-trip delay time from each host to its neighbors. The measurements are transmitted to the coordinator, where a corresponding strategy triggers an adaptation when the latency for a query exceeds the threshold specified in the query.

3) Language–Runtime Interface: To keep the ADAPTIVECEP language independent of the runtime, our implementation of the language is parametrised over an interface that defines the underlying runtime. For example, `run` (Figure 3, Line 12) takes a reference to the system runtime as argument. Crucially, the argument does not need to be given explicitly, but it can be inferred using Scala's implicit argument resolution. By declaring an implicit system value `implicit val sys: System = AkkaEsperBacked()`, the `run` method uses the specified system as back-end. This way, the user-code configuration for the System `sys` is decoupled from the queries it executes, i.e., configurations with different monitoring services and strategies.

VI. EVALUATION

The goal of our evaluation is to assess the ability of a CEP system implemented with ADAPTIVECEP to satisfy the demands set by the user upon changing conditions.

A. Evaluation Setup

We simulate the execution of a CEP system constituted of 25 fully-connected hosts which can exchange events over their connections. In the experimental setup, we model variability in environmental conditions (e.g., changing quality of data connection in a mobile device) by randomly changing latency and bandwidth of connections over time rather than using a full-fledged network simulator. Event producers and consumers are placed on fixed hosts. The adaptation process dynamically changes the placement of intermediate operators based on the activated strategy. We now present the simulation scenarios.

Static vs. Adaptive case: We consider two kinds of simulations, *Static* and *Adaptive*. Both simulations start with the same initial configuration. In the static case, operator placement is fixed. In the adaptive simulation, the strategy checks the current quality measurements on a regular basis and reconfigures the system if the quality demands are not satisfied by finding an operator placement that fulfills the demands. If the system is about to cross the defined threshold, the strategy compares the current configuration against a set of potential new configurations, where operators are relocated to other hosts, and adapts the system to the configuration which performs best under the given demand. Some adaptations may improve a performance measure at the detriment of another one. In such case, the system assumes it only has to optimize for the demand constrained by the user in the query. In case multiple demands are specified, we implemented a strategy that balances between both quality demands in case of conflicts by choosing a new configuration which maximizes the quality gains as compared to the current configuration. We use a metric which assigns equal weight to the proportional changes of all quality measurements when going from the current to a potential new configuration. For example, a possible new configuration which doubles bandwidth but also doubles latency compared to the current configuration will be considered neither gainful nor involving losses. For different use cases, strategies with other cost metrics may be more suitable. Resolution of conflicting demands opens a wide design space for strategies.

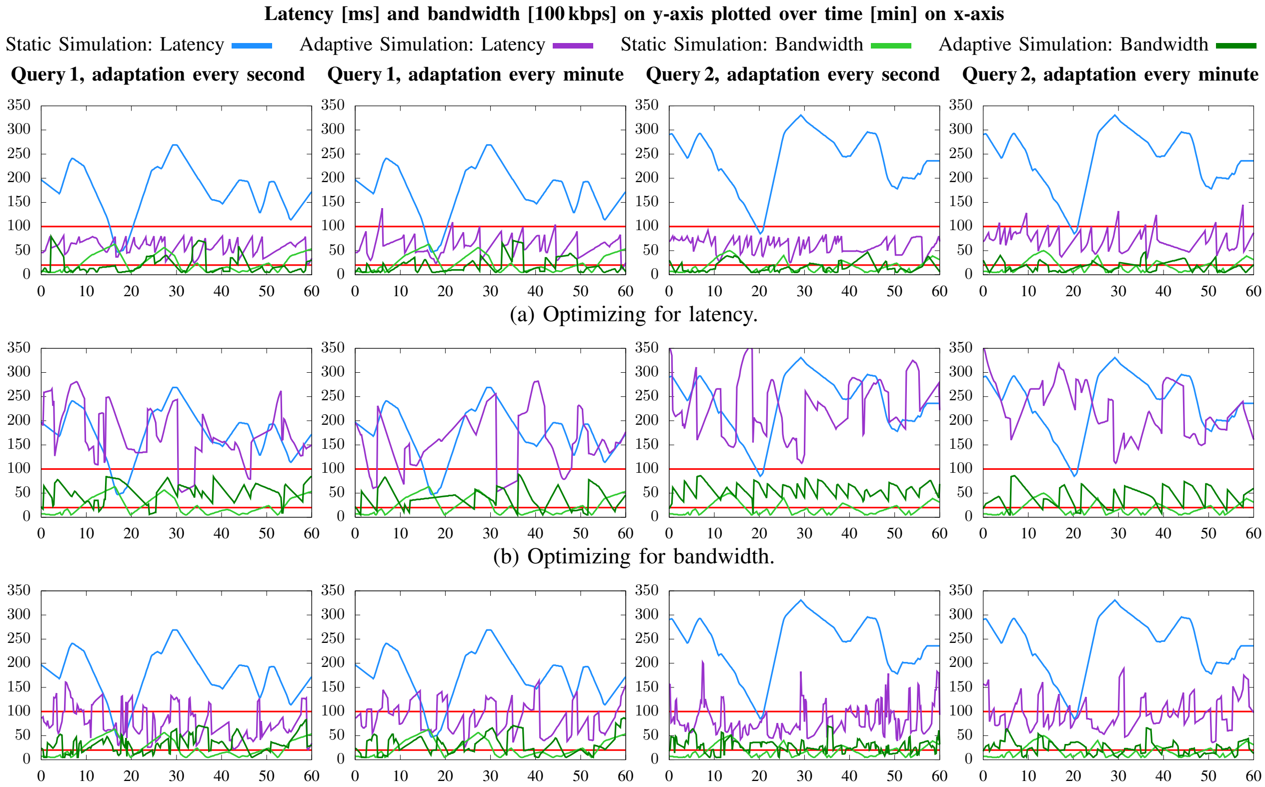
Demands: We consider two demands, latency and bandwidth, because they are crucial in online mobile scenarios for interactive multi-user applications. In the simulation, their values change randomly over time, modifying the quality properties of communication links in the simulated infrastructure.

Queries: We consider two queries. The first query used for the simulation has a structure similar to the example in Figure 3. We simulate one consumer and two producers. The event streams from the producers are joined by one operator and filtered by a second one. The second query is more complex, extending the first query by joining its result with the stream of another event producer.

Adaptation Rate: We consider two adaptation rates. In the High adaptation rate, adaptation occurs every second. In the Low adaptation rate, it occurs only once a minute.

B. Evaluation Results

Figure 9 shows the simulation results. Each plot reports the latency (blue lines) and bandwidth (green lines) measurements for both the static and the adaptive approach. The adaptive approach aims to keep latency *below* the latency threshold



(a) Optimizing for latency.

(b) Optimizing for bandwidth.

(c) Optimizing for both latency and bandwidth.

Figure 9: Evaluation results.

(upper red horizontal bar) and bandwidth *above* the bandwidth threshold (lower red horizontal bar). The first and second column display the results for Query 1, the third and fourth column for Query 2. The first and third column show the high adaptation rate for the adaptive simulation, the second and fourth column show the low adaptation rate. The static simulation looks the same in the first two and the last two columns, respectively, since it does not adapt to changing conditions.

Figure 9a shows the results for a given latency demand. For the adaptive strategy, the demand is never violated during the simulation, i.e., it stays below the threshold (upper horizontal bar) for both queries. In the non-adaptive static case, the latency is distinctly higher, exceeding the threshold for the most part of the simulation time. With the low adaptation rate, the latency demand is violated in a few cases before the system reconfigures itself. Figure 9b shows the system behavior in case of a bandwidth demand. Some adaptations performed to increase bandwidth actually result in higher latency. The system assumes it can make such trade-offs because the user defined a bandwidth but not a latency demand. The system is especially successful in ensuring the bandwidth demand for the second query using a high adaptation rate. Also, in the other three cases, the adaptive simulation regularly adapts to improve bandwidth (steep increases), which keeps it above the threshold compared to the static case, where bandwidth stays below the threshold for long periods of time. Figure 9c shows the results for users specifying demands on both latency and bandwidth. For such a multi-objective scenario, it is often impossible to satisfy both demands at the same time and more demand violations will occur. Also, quality of service for each demand does not decrease to the same extent as seen in Figure 9a and 9b, where the strategy is allowed sacrificing one demand for the other.

The simulation shows that system adaptations allow meeting the specified quality demands and quality awareness can be used to improve the system's performance. By providing language abstractions to the developer for specifying important demand characteristics for an application, AdaptiveCEP can adapt itself to satisfy the given demands.

VII. Related Work

The work in this paper spans over different areas. We consider related work in CEP and operator placement. We also provide an overview of FRP, which we extensively use in the design of our API and we consider previous (non-CEP) work on language-level adaptation.

A. Languages for CEP

As CEP deals with queries over streams of data, researchers have investigated dedicated languages to express those queries. CEP languages offer means to define event streams and operators to express their combination and correlation.

Event streams in CEP can be thought of as time-changing database tables. Queries are evaluated every time a new event occurs, i.e., a new entry in the table is added. Not surprisingly, languages for CEP have similarities with query languages for databases, e.g., SQL. Hence, they also share many limitations of database query languages, most notably, CEP queries are usually specified as strings, like in Esper [1], SASE [46], Cayuga [15] and TESLA [12]. As such, they do not take advantage of language integration, including safety guarantees from the compiler and integration with other language abstractions such as inheritance or late binding.

Recent languages for real-time analytics, like Microsoft Trill [8] for C#, make an attempt at language integration and are designed as embedded DSL in mainstream languages. Similarly, macros have been used to embed SQL-like expressions into Scala [3]. The recent introduction of lambdas in Java 8 allows to define queries over streams in a declarative way using pure Java syntax. Similar to our case, in Spark Streaming [47] developers express the event processing logic in a Scala DSL.

Yet, a fundamental limitation of existing languages for CEP, including all those mentioned above, is that they capture only a single aspect of the system: the definition of event combination and correlation. None of these languages takes into account elements like quality demands or conditions nor they provide an adaptation mechanism to fulfill those demands. In fact they provide *CEP operators* but no quality operators.

B. *Operator Placement*

Finding the optimal placement according to the performance and cost models is known as the operator placement problem. Existing work involves identifying an appropriate host to place an operator [35], [4] based on metrics like throughput and latency, but it does not take into account other constraints that are important in real-world systems, such as spatial location.

In the context of event-based communication, techniques exists to achieve low latency [44], to minimize the utilized bandwidth [43], to achieve reliable detection and delivery of events [24], or to achieve confidential transmission and ensuring privacy [45]. In MCEP [32], the delivered events are based on a range query whose focal point is the current location of the mobile device. The system ensures that the operator graph can always be mapped to new producers of information.

Though a number of operator placement algorithms are dynamic, i.e., exploiting runtime information to guide placement, to the best of our knowledge, no existing approach allows to select specific demands in the CEP language nor it is open to the implementation of new strategies like AdaptiveCEP.

C. *Functional Reactive Programming*

FRP [18] allows to define data flows declaratively and concisely via time-changing values. The runtime of the language automatically propagates the changes to dependent values. Over time a number of variants have been proposed. Bainomugisha et al. [5] provide an overview of the existing solutions.

Flapjax [28] introduced the use of FRP in Web applications, which has recently inspired a number of reactive libraries such as Rx.JS [27] and Bacon.js [34]. REScala [41] integrates event streams and time-changing values with object-oriented abstractions. Its runtime provides an event propagation system supporting dynamic dependencies for adding event queries during the execution. i3QL [29] adopts techniques from relational algebra and language-level optimizations to speed up event processing and enable incremental computation.

Recent research on RP focused on different propagation strategies to achieve properties such as glitch avoidance [11], efficient propagation over remote network connections [17] or concurrent propagation in a Web environment [13].

D. *Languages for Adaptive Systems*

Context-oriented programming (COP) [39] has been suggested to implement adaptive systems based on the MAPE-K autonomic computing model [40]. In this paradigm, the programming language provides *layers*, dedicated abstractions to represent behaviors that can be composed based on the change of external conditions. EventCJ [21] allows to control layer activation based on user-defined events and their combination.

SCEL [30] is a core language inspired by process calculi with a formal semantic foundation to reason about autonomic systems behavior. It provides dedicated abstractions to model behaviors, knowledge and aggregation based on policies, and to support context-awareness, self-awareness and adaptation. Degano et al. [14] propose a COP language which features the separation into a declarative part for the context and a functional part for actual computing. The formal approach allows to statically verify correctness properties for adaptations based on a type and effect system. S-CLAIM [6] is a declarative agent-oriented language for developing reactive mobile agents to achieve ambient intelligence and context-sensitivity.

Despite events being supported in some form by many of these languages (e.g., messages exchanged by processes, event conditions to trigger an adaptation, messages in agent communication), none of them specifically targets CEP.

VIII. Conclusion and Future Work

In this paper, we presented AdaptiveCEP, a novel CEP language and system that allows developers to specify not only event processing operators, but also quality demands that the system should fulfill. AdaptiveCEP supports programmable dynamic adaptation via strategies that implement the actual adaptation plan. An interface based on FRP allows to access and modify the current system configuration. Our evaluation shows that AdaptiveCEP allows to satisfy quality demands despite dynamic changes in the execution environment.

We are currently working on extending AdaptiveCEP to support more operators, e.g., ones that express complex movement patterns of event sources and sinks. Our vision is that such operators will allow not only to adapt the systems' behavior when necessary, but also to proactively trigger an adaptation when a demand violation is foreseen.

IX. Acknowledgments

This work has been co-funded by the German Research Foundation (DFG) as part of project C2 within the Collaborative Research Center (CRC) 1053 – MAKI, by the European Research Council, grant No. 321217, and by the LOEWE initiative (Hessen, Germany) within the NICER project.

References

[1] EsperTech – Esper. http://www.espertech.com/esper/, 2006. Accessed 05-01-2017.

[2] Akka. http://akka.io/, 2009. Accessed 05-01-2017.

[3] Slick (Scala Language Integrated Connection Kit). http://slick.typesafe.com/, 2012. Accessed 05-01-2017.

[4] Y. Ahmad and U. Çetintemel. Network-aware query processing for stream-based applications. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 456–467. VLDB Endowment, 2004.

[5] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A Survey on Reactive Programming. *ACM Computing Surveys*, 45(4):52:1–52:34, Aug. 2013.

[6] V. Baljak, M. T. Benea, A. E. F. Seghrouchni, C. Herpson, S. Honiden, T. T. N. Nguyen, A. Olaru, R. Shimizu, K. Tei, and S. Toriumi. S-CLAIM: An Agent-based Programming Language for Ami, A Smart-Room Case Study. *Procedia Computer Science*, 10:30 – 37, 2012.

[7] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Software Engineering for Self-Adaptive Systems. chapter Engineering Self-Adaptive Systems Through Feedback Loops, pages 48–70. Springer-Verlag, Berlin, Heidelberg, 2009.

[8] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, Dec. 2014.

[9] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[10] Cisco. Cisco Visual Networking Index: Forecast and Methodology. Technical report, 2005-2020.

[11] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP*, pages 294–308, 2006.

[12] G. Cugola and A. Margara. TESLA: A formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 50–61, New York, NY, USA, 2010. ACM.

[13] E. Czaplicki and S. Chong. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, New York, NY, USA, 2013. ACM.

[14] P. Degano, G.-L. Ferrari, and L. Galletta. A Two-Component Language for Adaptation: Design, Semantics and Program Analysis. *IEEE Transactions on Software Engineering*, 42(6):505–529, June 2016.

[15] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proceedings of the 10th International Conference on Advances in Database Technology*, EDBT'06, pages 627–644, Berlin, Heidelberg, 2006. Springer-Verlag.

[16] A. Diaconescu, Y. Maurel, and P. Lalanda. Autonomic Management via Dynamic Combinations of Reusable Strategies. In *Proceedings of the 2Nd International Conference on Autonomic Computing and Communication Systems*, Autonomics '08, pages 16:1–16:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[17] J. Drechsler, G. Salvaneschi, R. Mogk, and M. Mezini. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 361–376, New York, NY, USA, 2014. ACM.

[18] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM.

[19] A. Hinze, K. Sachs, and A. Buchmann. Event-based Applications and Enabling Technologies. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 1–15. ACM, 2009.

[20] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[21] T. Kamina, T. Aotani, and H. Masuhara. EventCJ: A Context-oriented Programming Language with Declarative Event-based Context Transition. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 253–264, New York, NY, USA, 2011. ACM.

[22] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[23] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107, New York, NY, USA, 2004. ACM.

[24] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, and M. Völz. Rollback-recovery without checkpoints in distributed event processing systems. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 27–38, New York, NY, USA, 2013. ACM.

[25] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 335–346, New York, NY, USA, 2008. ACM.

[26] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, July 2004.

[27] E. Meijer. Reactive extensions (Rx): Curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 11:1–11:1, New York, NY, USA, 2010. ACM.

[28] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM.

[29] R. Mitschke, S. Erdweg, M. Köhler, M. Mezini, and G. Salvaneschi. i3QL: Language-integrated Live Data Views. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 417–432, New York, NY, USA, 2014. ACM.

[30] R. D. Nicola, M. Loreti, R. Pugliese, and F. Tiezzi. A Formal Approach to Autonomic Systems Programming: The SCEL Language. *ACM Transactions on Autonomous and Adaptive Systems*, 9(2):7:1–7:29, July 2014.

[31] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. Artima Incorporation, USA, 2nd edition, 2011.

[32] B. Ottenwälder, B. Koldehofe, K. Rothermel, K. Hong, D. Lillethun, and U. Ramachandran. MCEP: A mobility-aware complex event processing system. *ACM Transactions on Internet Technology*, 14(1):6:1–6:24, Aug. 2014.

[33] B. Ottenwälder, B. Koldehofe, K. Rothermel, and U. Ramachandran. MigCEP: Operator migration for mobility driven distributed complex event processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 183–194, New York, NY, USA, 2013. ACM.

[34] J. Paananen. Bacon.js. http://baconjs.github.io/, 2012. Accessed 05-01-2017.

[35] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 49–, Washington, DC, USA, 2006. IEEE Computer Society.

[36] S. Rizou, F. Durr, and K. Rothermel. Solving the multi-operator placement problem in large-scale operator networks. In *Proceedings of 19th International Conference on Computer Communications and Networks*, pages 1–6, Aug 2010.

[37] M. Sabin. Shapeless. http://github.com/milessabin/shapeless, 2011. Accessed 05-01-2017.

[38] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):14:1–14:42, May 2009.

[39] G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-oriented Programming: A Software Engineering Perspective. *Journal of Systems and Software*, 85(8):1801–1817, Aug. 2012.

[40] G. Salvaneschi, C. Ghezzi, and M. Pradella. ContextErlang. *Science of Computer Programming*, 102(C):20–43, May 2015.

[41] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 25–36, New York, NY, USA, 2014. ACM.

[42] Y. Shoham. Software agents. chapter An Overview of Agent-oriented Programming, pages 271–290. MIT Press, Cambridge, MA, USA, 1997.

[43] M. A. Tariq, B. Koldehofe, G. G. Koch, and K. Rothermel. Distributed spectral cluster management: A method for building dynamic publish/subscribe systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 213–224, New York, NY, USA, 2012. ACM.

[44] M. A. Tariq, B. Koldehofe, and K. Rothermel. Efficient content-based routing with network topology inference. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 51–62, New York, NY, USA, 2013. ACM.

[45] M. A. Tariq, B. Koldehofe, and K. Rothermel. Securing Broker-Less Publish/Subscribe Systems Using Identity-Based Encryption. *IEEE Transactions on Parallel and Distributed Systems*, 25(2):518–528, 2014.

[46] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 407–418, New York, NY, USA, 2006. ACM.

[47] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.