Programming with Implicit Flows

Guido Salvaneschi, Technische Universität Darmstadt

Mira Mezini, Technische Universität Darmstadt and Lancaster University

Patrick Eugster, Purdue University and Technische Universität Darmstadt

// Modern software increasingly processes massive amounts of data, which has led to the emergence of advanced programming models. In these models, the software engineer declaratively defines computations as compositions of other computations without explicitly modeling how the data should flow along dependency relations, letting the runtime automatically manage and optimize data flows. //

WE'VE COME A LONG WAY from painstakingly feeding problem datasets into computer systems via punch cards. Computer systems have become much more convenient to interact with and are able to process much larger datasets, which are often kept in large-scale storage systems. However, computer systems are also much more commonly involved in processing data that's produced or modified online as the program executes, sometimes perpetually. This is particularly the case for applications specifically developed to react to real-world events, such as temperature changes or other environmental cues captured through sensors.

The last decade has thus seen the advent of abstractions and paradigms that support the development of *reactive* software. Central to such approaches is the concept of events that capture the dynamic occurrences that trigger computations. Over the years, several steps have been made in this direction, including language-level support for events, continuous time-changing values (signals or behaviors), constraints, asynchronous execution, and futures. The ever-increasing complexity of reactive applications has recently raised new interest around these abstractions.

The new paradigm of reactive programming focuses on a more holistic view that demands seamless integration of existing solutions, including constraints resolution to enforce functional dependencies, automatic updates of dependent values, and interoperability among different reactive abstractions such as signals and event streams. The goal is to raise the abstraction level: rather than explicitly reifying events in the software, changes to variable values are detected and propagated through programs by re-computing the values of all dependent variables implicitly during runtime. Interestingly, a similar trend can be observed in recent big data analysis software. Not too long ago, such programs were typically perceived as resembling complex queries applied to very large yet static datasets.

Researchers and practitioners have proposed a host of programming languages and models for such programs, which tend to mix imperative and declarative traits to expose the order of a non-cyclic computation network and are centered on some form of data structure conceptualizing the current state of

0740-7459/14/$31.00 © 2014 IEEE

computation. Despite improvements in running time of such analysis programs, their execution can still take sufficiently long to make repeated complete executions of the same program upon additions or changes to the underlying datasets prohibitively expensive. Consequently, recent improvements consist of enabling incremental computations—that is, reexecuting only those parts of queries that become invalid or incomplete by dataset changes.

Although reactive and big data analysis applications have little in common at first glance, we observe a shared trend in their respective programming models: they strive to capture *what* the computation ought to do but not *when* (or how) because the data is subject to computation changes over time (thus we speak of "data flows"). The execution engines and language runtimes increasingly carry the burden of determining which parts of computations are affected by which fluctuations in the processed data. As it's unlikely that runtime systems can determine these things entirely on their own—at least efficiently—or that such transparency would even serve the programmer, new abstractions are needed to capture such implicit flows in addition to underlying runtime support.

Events and Reactive Programming

Events are a common way for programmers to reason about significant conditions in both the environment and the program's execution. Mainstream languages have supported dedicated abstractions for events for a long time. For example, in C# events are class attributes that belong to the class's interface, in addition to methods and fields. Over the last few years, researchers have proposed increasingly sophisticated event models; see the "Advanced Programming with Events" sidebar for examples.

Integration into the object-oriented (OO) programming model has been enhanced to extend OO concepts such as inheritance to events and event handling. Early approaches like Java$_P$S implemented events as specific objects.[1] In EScala, events are first-class entities: as in C#, they're object attributes, just like methods and fields, and their definition is subject to polymorphic access and late binding.[2] Our investigations show that this is highly valuable, enabling programmers to encode a class's behavior as a state machine and extend it at this high level of abstraction rather than at the level of individual methods.[3]

Events in isolation improve little over the observer design pattern.

Events in isolation improve little over the observer design pattern. The difference becomes crucial when expressive operators for event combination are available to correlate events to define new (complex) events that capture high-level situations of interest. Advanced systems support operators to combine events with increasing levels of expressiveness. For example, the $e_1||e_2$ expression in EScala returns an event that fires when either e_1 or e_2 fire. Full-fledged embeddings of complex event processing such EventJava,[4] or stream-processing languages such as SPL,[5] support complex queries over event streams, including time windows and joins.

In parallel to the development of richer event models, other researchers have focused on more inherent data-flow and change-driven solutions for reactive applications. These approaches have old roots. For example, the Garnet and Amulet graphical toolkits support automatic constraint resolution to relieve the programmer from manual updates of the view.[6] In functional reactive programming (FRP), developers specify the functional dependencies among time-changing values in a reactive application, and the language runtime is responsible for performing the necessary updates (see the "Reactive Programming and Languages" sidebar).[7] FRP was developed in the strict functional language Haskell and initially applied to graphical animations; to date, researchers have applied it to several fields, including robotics and wireless sensor networks.

The fundamental concept in reactive languages is that programmers don't directly handle the control flow—rather, execution is driven by the implicit flow of data and the need to update values. Programmers specify constraints that express functional dependencies among values in the application, and the language runtime enforces these constraints without any further effort on the programmer's part.

More recently, these approaches have inspired many embeddings of

ADVANCED PROGRAMMING WITH EVENTS

Event-based languages include Join Java,[1] which captures events by specific asynchronous methods and supports joining of multiple events, and Ptolemy,[2] which supports features known from aspect-oriented programming (AOP).[3] In AOP, advices are triggered at points in the program's execution (for example, the end of a method call) that are referred to as join points, which can be seen as events that occur during the execution and treated uniformly with other events. For example, EScala **before(method)** and **after(method)** events are triggered before and after a method's execution. Also, in event-based languages that integrate AOP features, programmers can refer to all events of a certain type, a feature that resembles AOP quantification. As an example of an expressive event system, look at the following slice of a drawing application in EScala:

```
abstract class Figure { ...
    protected evt moved[Unit] = after(moveBy)
    evt resized[Unit]
    evt changed[Unit] = resized || moved || after(setColor)
    evt invalidated[Rectangle] = changed.map(() => getBounds())
   ...
    def moveBy(dx: Int, dy: Int) { position.move(dx, dy) }
    def setColor(col: Color) { color = col }
    def getBounds(): Rectangle
   ...
}
class Rectangle extends Figure {
    evt resized[Unit] = after(resize) || after(setBounds)
    override evt moved[Unit] = super.moved || after(setBounds)
   ...
    def resize(size: Size) { this.size = size }
    def setBounds(x1: Int, y1: Int, x2: Int, y2: Int) { ... }
}
```

Implicit events, such as **after(moveBy)** in the **Figure** class, are automatically triggered at the end of the associated method's execution (**moveBy**, in this case). Events can be defined declaratively by event expressions: the event **changed** is triggered when one of the events **resized**, **moved**, or **after(setColor)** is triggered. EScala events integrate with objects in several ways. Events support visibility modifiers, and abstract events, such as **resized**, can be refined in subclasses. Events can be overridden in subclasses (such as **moved**), and the inherited definitions can be accessed by **super**. Events are late-bound: in the expression **f.changed**, the definition of changed in **Figure** or in **Rectangle** can be picked up depending on the dynamic type of **f**.

JEScala extends EScala to include asynchronous events and joins such as Join Java and EventJava.[4] Join expressions fire an event after two or more events combined by & occur in any order. Multiple joins can be combined in *disjunctions* using the | operator; when multiple joins fire inside the same disjunction, one is chosen non-deterministically. Joins offer an alternative to thread-based concurrency. In the following Actor example, messages are asynchronous events (Lines 2-3); a disjunction (Line 9) ensures that a single message is processed at a time:

```
1  class Actor {
2      async evt helloMsg[Unit] = ...
3      async evt byeMsg[Unit] = ...
4
5      sync evt threadReady[Unit]
6      async evt start[Unit]
7      start += {while(true){threadReady()}}
8
9      evt (doHelloMsg,doByeMsg) = (threadReady & helloMsg)
10                                | (threadReady & byeMsg)
11     doHelloMsg += { println("Hello") }
12     doByeMsg += ...
13 }
```

References

1. S.V. Itzstein and D. Kearney, "The Expression of Common Concurrency Patterns in Join Java," *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications*, 2004, pp. 1021–1025.
2. H. Rajan and G.T. Leavens, "Ptolemy: A Language with Quantified, Typed Events," *Proc. 22nd European Conf. Object-Oriented Programming*, 2008, pp. 155–179.
3. G. Kiczales et al., "Aspect-Oriented Programming," *Proc. 11th European Conf. Object-Oriented Programming*, 1997, pp. 220–242.
4. J.M. Van Ham et al., "JEScala: e Modular Coordination with Declarative Events and Joins," *Proc. 13th Int'l Conf. Modularity*, 2014, pp. 205–216.

domain-specific languages (DSLs) and functional constraints in existing (imperative) programming languages. The advantage of this solution is that programmers specify a functional dependency in an intuitive, declarative way. Consequently, reactions are directly expressed, don't need to be inferred from the control flow, and can be easily composed.

In practice, continuous time-changing values—also known as signals—aren't enough. The need for events (discrete time-changing values) is explained by two observations:

- Events come from external phenomena that are inherently discrete, such as an interrupt or new data from a sensor.
- Events are better suited for modeling certain behaviors. In principle, a mouse click can be modeled as a Boolean continuous time-changing value that switches to true when the mouse is clicked, but most programmers would rather think of a mouse click as an event. For this reason, existing reactive languages provide both signals and events.

Reactive programming is an emerging trend, and identifying the boundaries of this field is hard. However, the following principles seem valid in general:

- *Declarative style*. Reactive behavior is defined in a direct, convenient, declarative style instead of encoding it in design patterns or through imperative updates of program state. Reactions are directly expressed and don't need to be encoded into the program's control flow.
- *Composition*. Abstractions allow for composition of more complex reactions. Traditional OO applications express reactions in callbacks that are executed when an observable changes. However, callbacks typically have side effects that modify the application's state but don't return a value. Consequently, they're hard to combine. Instead, events can be combined through combinators, and signals can be combined directly into more complex reactive expressions.
- *Automation*. Programmer effort is reduced by delegating the responsibility of reacting to changes in program state and updating corresponding entities to language runtime. This solution has several advantages. Reactive code is less error-prone because programmers don't forget to update dependencies (which introduce inconsistencies) and don't update defensively, independently of necessity (which wastes computational resources). In addition, automation enables optimization and more automated memory management.
- *Interoperability*. Different reactive abstractions can interoperate. Converting events into signals and back has an important role in practice. Several existing OO applications model state as object fields that are imperatively updated. Conversions allow programmers to take advantage of the design based on signals while still preserving compatibility with the existing nonfunctional code and the event-based design of many applications.

These principles—centered on the concept of implicit flows—highlight a significant similarity between reactive programming and big data analysis. The similarities between the two domains open perspectives for software that combine both paradigms.

Reactive programming is an emerging trend, and identifying the boundaries of this field is hard.

Big Data Analysis

Technologies spearheaded by Google's efforts such as the Google file system (GFS) or the distributed implementation of the MapReduce framework originally introduced in the Lisp programming language have ushered in a new era of scalable computing.[8] Through Apache's open source versions of such systems, bundled under the name Hadoop, these technologies have become widely available; they're currently considered part of the standard toolkit for programming with big data. GFS and the Hadoop distributed file system (HDFS) achieve scalability essentially by restricting write operations on files from arbitrary updates to append-only writes. HDFS serves as the default storage medium for data handled by Hadoop MapReduce or for results created by the same. With a distributed file system used between MapReduce tasks, multiple individual local disks used between the map and reduce phases of such tasks, and several mappers and reducers splitting the workload, the MapReduce toolchain can scale to very large input files.

To ease the burden on programmers, several high-level scripting and programming languages and language extensions have been introduced, exposing data flow to parallelization. They view programs as directed acyclic graphs (DAGs), with edges representing the flow of data and nodes representing (sets

REACTIVE PROGRAMMING AND LANGUAGES

Reactive programming is based on constraints enforced by the runtime. Consider a functional dependency among the variables a, b, and c such that a = b + c:

```
1  a = 2
2  b = 3
3  c = a + b
4  a = 4 // c is still 5
5  c = a + b // c = 7
```

```
1  a = 2
2  b = 3
3  c := a + b // Constraint
4  a = 4 // c = 7
```

In imperative programming (left), the functional dependency is true only immediately after the execution of the statement in Line 3. As soon as a change occurs, the functional dependency is no longer valid and must be updated manually (Line 5). Reactive languages (right) automatically enforce constraints (Line 3), recomputing functional dependencies when they aren't valid anymore.

As an illustration of more explicit use of constraints, consider the following minimal GUI application in the REScala reactive language.[1] The application counts the number of mouse clicks on a button, displays the result, and changes the button label when counting starts. In REScala, signals express functional dependencies in a declarative style. The traditional design without reactive programming for such an application adopts the observer design pattern. An implementation (simplified for the presentation) using the Scala Swing libraries looks as follows:

```
1  /* Create the graphics */
2  title = "Reactive Swing App"
3  val button = new Button {
4    text = "Click me!"
5  }
6  val label = new Label {
7    text = "No button clicks registered"
8  }
9  contents = new BoxPanel(Orientation.Vertical) {
10   contents += button
11   contents += label
12 }
13 /* The logic */
14 listenTo(button)
15 var nClicks = 0
16 reactions += {
17   case ButtonClicked(b) =>
18     nClicks += 1
19     label.text = "Number of button clicks: " + nClicks
20     if (nClicks > 0)
21       button.text = "Click me again"
22 }
```

The previous code requires inspecting the whole control flow to understand the update logic. For example, the text over the button is initialized in Line 4 and assigned in the statement in Line 21, which is conditionally executed based on variable `nClicks`, modified in Line 18. In the reactive programming version using REScala, the whole update logic is captured in Lines 5-11:

of) operations involving data from their incoming edges and results being passed onto outgoing edges. Pig Latin, an untyped scripting language from Yahoo, is a popular example of such a language. Hadoop Pig implements it on top of Hadoop MapReduce.[9] Languages such as Pig Latin can express data analysis jobs across domains like science and engineering, business and finance, and government and defense. In corresponding programs, intermediate state is typically incarnated by various types of data structures or collections representing large datasets (see the "Programming with Big Data" sidebar).

In general, languages for big data analysis roughly build on two abstractions:

- *Data structures*. The state of a DAG-based computation at a particular point in the DAG consists of intermediate data, which is conceptualized by a data structure. Data constraints and characteristics (ordering, indexing) are captured through data structure choice (bag versus set, set versus associative map, and so on). Pig Latin, for example, leverages bags and maps, while others propose collections and tables.
- *Operations and functions*. Computation itself is expressed via operations more typical of relational query models (filter, group, join) or functions (max, min, avg), which are applied to data structures; results are typically represented as data structures.

When data analysis programs or subprograms are translated to MapReduce jobs, the actual data structures will never be incarnated as such in a given process's address space or even across several such address spaces; these data structures serve uniquely as conceptual abstractions.

```
title = "Reactive Swing App"
val label = new ReactiveLabel
val button = new ReactiveButton

val nClicks = button.clicked.count
label.text = Signal{
  (if (nClicks() == 0) "No"
  else nClicks()) + " button clicks registered" }
button.text = Signal{
  "Click me" + (if (nClicks() == 0) "!"
                 else " again ") }
contents = new BoxPanel(Orientation.Vertical) {
  contents += button
  contents += label
}
```

In reactive languages, conversions between signals and events assume great importance. Conversions let you introduce signal-based (declarative) code into object-oriented event-based applications, abstract over state, and concisely express reactive computations.

The following REScala code snippet uses the snapshot conversion function to combine a signal that holds the current mouse position and a click event from the mouse. As a result, the snapshot returns a signal that holds the position of the last mouse click. The other example demonstrates the last(n) function, which holds a list of the last n values associated to an event stream. Here, last(n) computes the average in a sliding window of five values over a stream of events carrying integers:

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

```
val e = new ImperativeEvent[Double]
val window = e.last(5)
val mean = Signal { window().sum / window().length }
mean.changed += { println(_) }
```

Other reactive languages include FrTime,[2] Flapjax,[3] and Scala.React.[4] Currently, reactive languages are being extended to support automated propagation of individual elements of nontrivial data structures (lists[5]) or to distribution of reactive values over many nodes.[6]

References

1. G. Salvaneschi, G. Hintz, and M. Mezini, "REScala: Bridging between Object-Oriented and Functional Style in Reactive Applications," *Proc. 13th Int'l Conf. Modularity*, 2014, pp. 25–36.
2. G.H. Cooper and S. Krishnamurthi, "Embedding Dynamic Dataflow in a Call-by-Value Language," *Proc. 15th European Conf. Programming Languages and Systems*, 2006, pp. 294–308.
3. L.A. Meyerovich et al., "Flapjax: A Programming Language for Ajax Applications," *Proc. 24th ACM SIGPLAN Conf. Object-Oriented Programming Systems Languages and Applications*, 2009, pp. 1–20.
4. I. Maier and M. Odersky, "Deprecating the Observer Pattern with Scala.react," EPFL-REPORT-176887, 2012.
5. I. Maier and M. Odersky, "Higher-Order Reactive Programming with Incremental Lists," *Proc. 27th European Conf. Object-Oriented Programming*, 2013, pp. 707–731.
6. G. Salvaneschi, J. Drechsler, and M. Mezini, "Towards Distributed Reactive Programming," *Proc. Int'l Conf. Coordination Models and Languages*, 2013, pp. 226–235.

Restricting big data analysis and processing to computations that can be represented as DAGs is a strong limitation. Two major extensions of the computational model promoted by MapReduce and its associated early high-level languages address this limitation:

- *Incremental computation*. Support for such computation avoids making changes to input re-executing entire programs. Incremental computation is particularly sensible in the context of big data—many applications operate on input datasets such as logs, client activity records, or user records that are constantly extended. Based on the append-only semantics for many such files (by virtue of the distributed file system), extensions to datasets are naturally captured through stratified appendages.
- *Iterative computation*. Support for cycles during computation allows for a far more expressive computing model and is especially relevant in big data processing, where due to sheer data size, "one-shot" solutions are impossible and computations are iterated until they converge satisfactorily. A popular example is Google's page rank for determining webpage popularity, which originally motivated MapReduce. Other examples include machine-learning algorithms such as logistic regression.

Based on these needs, recent programming models aim to support iterative or incremental computing, or both. To that end, datasets are kept in main memory and partitioned across the various nodes necessary to accommodate them, thus making cross-accesses for updates much faster than on stored files.

PROGRAMMING WITH BIG DATA

Several programming languages and models are similar in spirit to Pig Latin. FlumeJava is a library for data-flow processing in Java proposed by Google[1] and also implemented by Apache Crunch.[2] FlumeJava compiles corresponding tasks to MapReduce jobs at runtime. Like the early Dryad language[3] or Pig Latin, the model comes with standard operators for joining data flows but also supports application-defined functions. The following implements a simple word count in FlumeJava:

```
PCollection<String> lines =
    readTextFileCollection(input_file);
PCollection<String> words = lines.parallelDo(
    new LineToWordFunction<String, String>(),
    collectionOf(strings()));
PTable<String, Long> wordCounts = words.count();
wordCounts.write(output_file);
```

First the program reads the **input_file** as a text file, and then, with some degree of parallelization chosen by the runtime, parses lines, generating a collection of strings. Next the program creates a table indexed by words, with the counts for the respective words, before, finally, writing the table to **output_file**.

Early innovators in terms of incremental and iterative computation were the Incoop[4] and iHadoop[5] extensions of Hadoop, respectively. Recent examples of data-processing models supporting these two features by storing data in main memory include distributed arrays in Presto[6] or resilient distributed datasets in Spark.[7] Incremental computation is thus far not supported by FlumeJava or Crunch; in the word count example, incremental computation would consist of augmenting the word counts' output to **output_file** following the order of the program, upon extensions to **input_file**. With an in-memory representation of the **wordCounts** table, it would suffice to apply the previous stages to any lines added to the **input_file** and subsequently adding the corresponding new word counts to existing ones in **wordCounts**, or creating new entries to the table for words that previously weren't encountered.

References

1. C. Chambers et al., "FlumeJava: Easy, Efficient Data-Parallel Pipelines," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2010, pp. 363–375.
2. "Incubator Crunch," Apache Software Foundation, 2013; http://incubator.apache.org/projects/crunch.html.
3. M. Isard et al., "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *SIGOPS Operating System Rev.*, vol. 41, 2007, pp. 59–72.
4. P. Bhatotia et al., "Incoop: MapReduce for Incremental Computations," *Proc. 2nd ACM Symp. Cloud Computing*, 2011, article no. 7.
5. E. Elnikety, T. Elsayed, and H. Ramadan, "iHadoop: Asynchronous Iterations for MapReduce," *Proc. 3rd Int'l Conf. Cloud Computing Technology and Science*, 2011, pp. 81–90.
6. S. Venkataraman et al., "Using R for Iterative and Incremental Processing," *Proc. 4th Usenix Conf. Hot Topics in Cloud Computing*, 2012, p. 11.
7. M. Zaharia et al., "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-memory Cluster Computing," *Proc. 9th Usenix Conf. Networked Systems Design and Implementation*, 2012, p. 2.

Towards Unified Programming with Implicit Flows

The two families of programming languages and language extensions considered in the previous sections share a new paradigm for processing data: implicit data flows "through" computations. While the two thrusts currently still emphasize different settings and requirements—low-latency in-memory processing on one or a few nodes with small data volumes for reactive programming, and high throughput processing of large datasets distributed across many nodes for big data analysis—confluences are starting to emerge:

- Approaches in each family are extended with features characteristic of the other family. Specifically, implicit propagation of changes in reactive programming is generalized from simple values to data collections and from local to distributed computations; support for incremental and iterative computations is being added to big data analytics approaches.
- Approaches with uniform abstractions for processing heterogeneous stored and online data sources are emerging. The reactive extensions (Rx)[10] of .NET represent a library-based approach to modeling complex event and stream processing by LINQ operators,[11] which are also used for stored data processing; following DEDUCE,[12] Shark combines MapReduce,[13] designed for stored data analysis, with support for processing online data.

These first steps are promising, but there's a need for a much stronger confluence. We believe that modern applications would benefit from integrating time-changing values (signals and big data processing abstractions) and making them composable. To enable such compositions, we need to conciliate propagation of changes on both immutable data in the style of

FRP and mutable data characteristic of big data processing. Fine-grained changes over mutable data structures are an instance of a more general problem requiring further advances in incrementalization techniques. The database community has studied this for a long time under the label of view maintenance. More recently, researchers have applied incremental solutions to specific programming domains, such as incremental collections. However, attempts to incrementalize a generic program are just beginning. Beside incrementalization, language integration of uniform abstractions for implicit data flows could enable optimizations across data-flow graphs, offering opportunities to apply compiler optimizations such as inlining, partial evaluation and staging, loop fusion, and deforestation.

The itegration of reactive programming and big data analysis poses several challenges related to the composition of heterogeneous data management and processing strategies. It could require advanced module concepts and related type systems to express functionality that abstracts over a whole range of processing strategies as well as different data sources or sinks. A key challenge is to reconcile flexibility with static typing to reduce runtime errors. This aspect is especially important in the context of big data, where a failure can propagate across dependent computations and invalidate processing already performed.

Acknowledgments
This work has been supported by the German Federal Ministry of Education and Research (BMBF) under grant number 16BY1206E, the European Research Council, grant number 321217, the Alexander von Humboldt foundation, and the US Defense Advanced Research Projects Agency, grant number #N11AP20014.

ABOUT THE AUTHORS

GUIDO SALVANESCHI is a postdoctoral researcher at Technische Universität Darmstadt. He's interested in programming languages, reactive programming, event-based programming, and languages for adaptive systems. Salvaneschi received a PhD in computer science from Politecnico di Milano. Contact him at salvaneschi@cs.tu-darmstadt.de.

MIRA MEZINI is a professor of computer science at Technische Universität Darmstadt. Her research interests include programming languages and software development paradigms/tools, adaptable software architectures, software product-line engineering, and service-oriented architectures. Mezini received a PhD in computer science from the University of Siegen. She has served as the general and program chairs of several software engineering and programing language conferences and regularly serves on their program committees. Contact her at mezini@informatik.tu-darmstadt.te.

PATRICK EUGSTER is an associate professor in computer science at Purdue University, on leave for Technische Universität Darmstadt. He's interested in distributed systems and programming languages. Eugster received a PhD in computer science from EPFL. He's a recipient of the NSF Career Award (2007) and an ERC Consolidator Award (2012); he's also a member of DARPA's Computer Science Study Panel. Contact him at p@cs.purdue.edu.

References

1. P. Eugster and R. Guerraoui, "Distributed Programming with Typed Events," *IEEE Software*, vol. 21, no. 2, 2004, pp. 56–64.
2. V. Gasiunas et al., "EScala: Modular Event-Driven Object Interactions in Scala," *Proc. 10th Int'l Conf. Aspect-Oriented Software Development*, 2011, pp. 227–240.
3. G. Salvaneschi and M. Mezini, "Towards Reactive Programming for Object-Oriented Applications," *Trans. Aspect-Oriented Software Development XI*, LNCS 8400, Springer, 2014, pages 227–261.
4. P. Eugster and K. Jayaram, "EventJava: An Extension of Java for Event Correlation," *Proc. 23rd European Conf. Object-Oriented Programming*, 2009, pp. 570–594.
5. M. Hirzel et al., "IBM Streams Processing Language: Analyzing Big Data in Motion," *IBM J. Research and Development*, vol. 57, nos. 3/4, 2013, article no. 1.
6. B.A. Myers et al., "The Amulet Environment: New Models for Effective User Interface Software Development," *IEEE Trans. Software Eng.*, vol. 23, no. 6, 1997, pp. 347–365.
7. C. Elliott and P. Hudak, "Functional Reactive Animation," *Proc. 2nd ACM SIGPLAN Int'l Conf. Functional Programming*, 1997, pp. 263–273.
8. S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *Proc. 19th ACM Symp. Operating Systems Principles*, 2003, pp. 29–43.
9. C. Olston et al., "Pig Latin: A Not-so-Foreign Language for Data Processing," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, 2008, pp. 1099–1110.
10. J. Liberty and P. Betts, *Programming Reactive Extensions and LINQ*, 1st ed., Apress, 2011.
11. E. Meijer, B. Beckman, and G. Bierman, "LINQ: Reconciling Object, Relations and XML in the .Net Framework," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, 2006, p. 706.
12. V. Kumar et al., "DEDUCE: At the Intersection of MapReduce and Stream Processing," *Proc. 13th Int'l Conf. Extending Database Technology*, 2010, pp. 657–662.
13. R. Xin et al., "Shark: SQL and Rich Analytics at Scale," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, 2013, pp. 13–24.