Ways to React: Comparing Reactive Languages and Complex Event Processing

Alessandro Margara
High Performance Distributed Computing Group
Vrije Universiteit Amsterdam
a.margara@vu.nl

Guido Salvaneschi
Software Technology Group
Technische Universität Darmstadt
salvaneschi@informatik.tu-darmstadt.de

ABSTRACT

Reactive applications demand for detecting the changes that occur in a domain of interest and for timely reactions. Examples range from simple interactive applications to complex monitoring tasks involving distributed and heterogeneous systems.

Over the last years, different programming paradigms and solutions have been proposed to support such applications. In this paper, we focus on two prominent approaches: event-based programming, specifically Complex Event Processing (CEP), and Reactive Languages (RLs).

CEP systems enable the definition of high level situations of interest from low level primitive events detected in the external environment. On the other hand, RLs support time-changing values and their composition as dedicated language abstractions. These research fields have been investigated by different communities, belonging respectively to the database and the distributed systems areas and to the programming language area.

It is our belief that a deeper understanding of these research fields, including their benefits and limitations, their similarities and differences, could drive further developments in supporting reactive applications. For this reason, we propose a first comparison of the two fields. Despite huge differences, we believe that such a comparison can trigger an interesting discussion across the communities, favor knowledge sharing, and let new ideas emerge.

Keywords

Reactive Applications, Reactive Programming, Complex Event Processing, Event-Driven Programming

1. INTRODUCTION

Reactive applications respond to the occurrence of events of interest —e.g., user interaction or changes in the state of components— by performing some computation, which may in turn trigger new events.

Designing, implementing, and maintaining reactive applications is difficult. First of all, applications may be interested in detecting (and reacting to) situations that are hard to identify, and require observing, collecting, and reasoning over multiple events. Second, low response time is often a key requirement, demanding for efficient algorithms and implementation strategies for event detection and reaction. Additionally, reactive code is asynchronously triggered by event occurrences, which makes it difficult to understand the control flow of the application. This problem is exacerbated in parallel and distributed scenarios, which involve synchronization issues.

The problem of supporting reactive applications has been tackled from different standpoints and by different communities, with heterogeneous backgrounds and expertise. Two prominent solutions emerged: event-based programming —particularly Complex Event Processing (CEP) systems— and Reactive Languages (RLs).

It is our belief that these two approaches are vastly complementary. For this reason, a deeper understanding of their strength, limitations, similarities, and differences can promote the integration of ideas and solutions coming from the two worlds and drive further advancements in the research on reactive applications.

This paper is a first attempt to compare the two fields, with the purpose of triggering the discussion across the communities and favoring knowledge exchange. At the same time, it proposes initial ideas for future research directions.

The paper is organized as follows: Section 2 briefly introduces the main features of CEP systems and RLs. Section 3 compares the two fields along various directions of analysis. Section 4 discusses open research challenges and introduces some initial proposals for future research. Finally, Section 5 surveys related work and Section 6 concludes the paper.

2. BACKGROUND

This section introduces CEP and RLs and highlights their key features.

2.1 Complex Event Processing

CEP [31, 19] is a form of information flow processing [15] specifically devoted to the definition and detection of high level situations of interest —or *composite* events— starting from low level *primitive* event notifications. Composite events are specified through user-defined queries, or *rules*, which express how to select, manipulate, and combine primitive events. As a consequence, the expressiveness of a CEP system depends on the language adopted for rule definition.

Differently from other information flow processing systems, which mainly adopt languages derived from SQL, CEP systems heavily rely on pattern detection [15]: rules define composite events starting from patterns of primitive ones, involving content-based and temporal constraints. Common to all CEP systems is the central role of time: events are timestamped, and sequences and time series are key components in every CEP language.

Nevertheless, significant differences exist among the proposed solutions: some of them exploit relatively simple languages, e.g., temporal extensions of regular expressions [7, 23]. Other systems introduce more powerful abstractions, e.g., based on logic programming [4] or on temporal logic [12, 13].

As an example, consider Rule R below, expressed in the TESLA language [12], which defines a `Fire` composite event starting from the observation of primitive events about the value of the temperature (`Temp`) and the presence of `Smoke` and `Rain` in a certain area.

```
Rule R
define  Fire(area: string)
from    Smoke(area=$a) and
        Avg(Temp(area=$a).value
        within 5 min. from Smoke) > 45 and
        not Rain(area=$a) within 10 min. from Smoke
where   area=Smoke.area
```

Rule R introduces most of the typical components of a CEP rule. It computes an aggregate value (average, `Avg`) over all the temperature readings in a given window of time (`5 min.`); it introduces a constraint on such value (`>45`); it uses a parameter (`$a`), requiring `Smoke` and `Temp` to be detected in the same area; it expresses a negation, demanding for an absence of `Rain`.

Researchers and practitioners in the CEP area have put significant effort on performance and scalability, defining efficient algorithms and evaluation mechanisms that enable high event throughput and low latency processing.

2.2 Reactive Languages

RLs provide dedicated abstractions to model time-changing values, usually referred to as *behaviors* [10] or *signals* [32]. This solution is considered superior to the Observer pattern, traditionally adopted in object oriented programming, which lacks of composability, inverts the logical relation among reactive entities and reduces the readability of applications [36].

Signals exhibit better composability and directly express the programmer's intention. They are defined by expressions that involve other signals and are updated automatically when a signal in the expression changes. As an example, consider the following code snippet, defined using the syntax of REScala [42].

```
Snippet T
val tick = new Var(0)
val hour = Signal{ tick() % 24 }
val day  = Signal{ (tick()/24) % 7 + 1 }
```

Snippet T controls the elapsing of time in a simulation. The value `tick` is a *var* in REScala terminology, i.e., an observable reactive value updated imperatively and used in signal expressions. The value `hour` is a signal that holds the elapsed hours in the simulation. It is defined by the signal expression `tick()%24` and depends on the value of `tick`. The value `day` is also a signal defined by the expression `(tick()/24)%7+1`. In the example, when `tick` changes, the values of `hour` and `day` are automatically recomputed by the reactive framework.

The applications of reactive programming include graphical animations, robotics, and wireless sensor networks. Reactive programming has been first proposed in purely functional languages —Functional Reactive Programming (FRP) [24]— and later introduced in a wider range of programming languages [36]. Current research includes the interaction with other language abstractions such as events, objects mutability, encapsulation and inheritance [42].

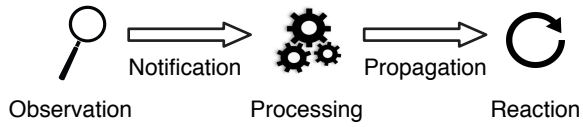

Figure 1: High Level View of a Reactive Application

3. COMPARING CEP AND RLS

To compare CEP and RLs, we start from a high level conceptual model of reactive applications —Section 3.1— and then we move to a more detailed analysis of the two approaches —Section 3.2—.

3.1 CEP and RLs from 10000 Feet

Figure 1 shows the high level view of a generic reactive application. We identify five main phases for the reactive behavior. First, a fact of interest is *observed* at some source. It is encoded into a *notification*, which triggers a computation in the following *processing* phase. The results of this computation are *propagated* to the interested components, which are responsible for *reacting* to them.

Table 1 compares the implementations of these phases in CEP and RLs. In CEP systems, sources observe and propagate generic events: typical examples are readings from sensor nodes in environmental monitoring, or from RFIDs in products lifecycle management, or stock values in financial analysis.

On the contrary, RLs are specifically designed to capture and propagate value changes of one or more variables. Primitive sources are vars or signals that are directly available in the reactive framework (a common example is the `time` signal, holding the current system time).

Event notifications are explicit in CEP and usually pushed from the sources to the component or components responsible for processing. Processing is performed according to a set of rules that predicate how to manipulate and combine explicit event notifications to produce the desired results (i.e., the composite events).

On the contrary, in RLs, notifications are implicit. Processing is defined through an expression that specifies how the output value is defined in terms of the input values. When writing an expression, users only consider input values, and notifications of value changes are fully transparent. As in CEP systems, most RLs adopt a push strategy to deliver event notifications to the processing component.

Similar to notifications, the propagation of results is explicit in CEP —implemented as the delivery of composite events— and implicit in RLs. In both cases, the communication is multicast to serve all the components interested in the results.

In CEP, the propagation phase is usually implemented adopting a push-based approach. On the contrary, some RLs delay the propagation (or even the computation) of results until some component needs to access them [32].

After delivering the results, CEP systems do not impose any limitation on the reactive phase. This phase is actually outside the scope of CEP: external clients receive composite events and can implement their own custom procedures for reacting to them. On the contrary, reaction is a key component of RLs and always determines a change into the value of a time-changing variable (signal).

3.2 The Devil is in the Details

This section provides a more detailed analysis of some key aspects of CEP and RLs.

3.2.1 Language Expressiveness

Both CEP systems and RLs offer declarative ways to define new entities: CEP rules define composite events from primitive ones,

Phase	CEP	RLs
Observation	Generic Events	Value Changes
Notification	Explicit – Push	Implicit – Push
Processing	Rules (from primitive to composite events)	Expressions (from signals to signals)
Propagation	Explicit – Multicast – Push	Implicit – Multicast – Push or Pull
Reaction	Generic Procedures – User-Defined	Value Changes

Table 1: A Comparison of how CEP and RLs Implement the Five Phases of a Reactive Behavior

RLs define time-changing values (signals) based on other signals.

We can observe a first difference in the nature of the input: in RLs, expressions consider signals that hold a value at any point in time and recompute the output upon changes. CEP rules consider events that occur at specific time points (or intervals): more specifically, they consider a time-annotated sequence (or *history*) of event occurrences and detect patterns of interest over it.

Similarly, in RLs the output of an expression is a new time-changing value. At each recomputation, the value is updated. On the contrary, in CEP systems, *new* composite events are generated over time: the output is not a single value, but a time-annotated history of composite events constantly updated with new occurrences.

Noticeably, time plays a central role in CEP systems. Not only input and output events are time-annotated, but time constraints and operators for sequence detection are first class entities in almost all CEP languages.

In RLs, less attention has been given to time. However, dedicated operators allow one to capture the history of a signal. For instance, in REScala [42], an expression like `a.last(n)` returns a list with the last `n` values of signal `a`. Another operator that works on time is `delay(n)`, which returns the value that signal `a` had `n` steps before.

3.2.2 Composability

One of the main interesting features of reactive programming is its support for composability. Signals can be composed into expressions that produce new signals. A fundamental property is that the expressions used to compose signals are not different from the expressions used to compose traditional values in the underlying language[1].

Similarly, most existing CEP systems enable the usage of composite events into rules that define other composite events. This promotes the creation of so called *hierarchies of events*, allowing the users to recursively abstract from low level details to define higher level concepts. Interestingly, there is usually no difference between rules that adopt only primitive events and rules that also consider composite events.

3.2.3 Consistency

In reactive languages, time-changing values are defined on top of other time-changing values. As an example, consider Expression E, which defines a signal `a` that depends on `b` and `c`:

Expression E
```
a = Signal{ b() + c() }
```

[1]In some reactive languages, in practice, differences can be observed. For example in Scala.React a signal expression requires the apply `()` operator on the signals that appear in a signal expression: the expression `Signal{a+b}` must be written as `Signal{a()+b()}`. However these aspects can be considered implementation details in case reactive systems is developed as library or as an embedded DSL. Languages like Flapjax overcome this limitation providing a dedicated compiler that automatically introduces the required syntactic sugar.

Let us assume that both `b` and `c` depend on another time-changing value, `d`. When `d` changes, the change should be propagated to `b` and `c` before updating `a`. Otherwise, there may be an interval of time in which `a` holds an invalid value, e.g., computed from the new value of `b` and the old value of `c`. This problem is known as *glitch* [10].

To ensure *glitch freedom*, RLs typically organize signals in levels and enforce a correct propagation order across levels. In our example, `a` needs to be at a higher level than `b` and `c`, which are at a higher level than `d`. In this way, `a` is updated only after both `b` and `c` have been updated.

In CEP systems, primitive events are usually processed in timestamp order and composite events are generated and propagated in timestamp order. Event notifications can be timestamped at the sources or at the processing engine, after the notification phase. Problems related to synchronization and out-of-order delivery are normally addressed before the processing phase, which simply assumes in-order arrival of events. Users typically lack any control on the order of evaluation of rules.

This solution defines a partial order among composite events. Indeed, there are usually no guarantees in the generation and propagation order of composite events having the same timestamp (concurrent events).

In case of hierarchies of events, which enable the occurrence of a composite event to trigger the occurrence of other composite events, rules get re-evaluated until a fixed point is reached, in which no further events are generated. This ensures that all the events having a timestamp t are successfully generated and propagated before any other event having a timestamp $t' > t$.

3.2.4 Performance

Performance is one of the main concerns of the researchers and practitioners working on CEP. Usually, the CEP service is implemented as a stand-alone component, which collects primitive events from sources and distributes derived composite events. For this reason, scalability in the rate of input events, in the number of deployed rules, and in the number of clients (event sources and receivers) becomes a key requirement.

Several proposals have been presented to reduce the processing delay and to increase the maximum throughput of CEP systems, including rule rewriting techniques [44], sharing of operators between multiple rules [7], and design of algorithms for parallel hardware architectures [34, 14, 40]. At the same time, industrial vendors advertise and compete on the performance of their solutions (see for example [18]).

So far, the community working on RLs has devoted less attention to performance issues and focused more on developing suitable programming abstractions and on promoting language integration. There are, however, attempts to optimize reactive languages. As an example, *lowering* was proposed as a technique to collapse several signal expressions into a single one [8].

Apart from that, most of the optimization effort has been carried on by the Functional Reactive Programming (FRP) commu-

nity. Nillson explored the use of Generalized Algebraic Data Types (GADTs) to dynamically optimize arrows-based embeddings of FRP in Haskell [37]. Other researchers proposed statically bounded time execution for real-time reactive programs [47] and compilation to efficient C code for applications written in the FRP style [48].

Typically, time-changing values are recomputed from scratch when a signal they depend on changes. In case of computationally intensive expressions, this may represent a waste of resources. For this reason, another solution to improve the performance of reactive languages is *incrementalization*. This technique applies when signals hold complex values, like objects.

Optimization mechanisms based on incremental and partial recomputation of values have been proposed for future research [43]. This may involve identifying —automatically or through user suggestions— efficient ways to recompute a value when only some of the values it depends on change.

Recent research integrated incrementalization of certain data structures into reactive languages [33]; however, applying this technique in the general case is till an open research challenge.

3.2.5 Distribution

Most CEP systems adopt a client/server architecture, where a CEP engine acts as a server, which collects and distributes events to the connected client components.

To improve scalability, several solutions have been proposed that distribute the processing load over multiple machines. Some of them require manual deployment of the processing tasks [2]; others offer automatic partitioning of rules into basic operators and provide an efficient deployment based on the user requirements and on the communication and load conditions [38, 11, 16].

The problem of finding a good deployment is referred to as the *operator placement* problem, which is known to be NP-hard [45]. Because of this, most of the solutions presented for the operator placement problem are based on heuristics or approximations.

In this context, existing approaches are extremely heterogeneous, and propose disparate solutions depending on the deployment scenario —e.g., cluster with fast connectivity or large scale distributed system— and on the user interests —e.g., minimize the processing latency rather than minimize network usage— [28].

Distribution has not been widely considered in RLs. An exception is AmbientTalk/R [30], a reactive language for pervasive and mobile applications. Apart from AmbientTalk/R, existing reactive languages have been applied to a distributed setting only as a programming solution for each single host. For example, Flapjax [36] is designed for the JavaScript components of a client/server application. However, none of the existing approaches considers a reaction that spans over the network, e.g., enforcing glitch-freedom across different hosts. An initial analysis of the problems connected with distributed reactive programming and a research roadmap have been first proposed in [41].

3.2.6 Safety

Signals are integrated in the language. As a result, in statically typed languages, signals are typechecked and the compiler ensures that *a well typed program cannot go wrong*. For example, the expressions that combine existing signals must be type safe.

CEP systems usually adopt simple data models. More in general, this is true for most information flow processing systems: indeed, as observed in [15], two main data models emerged.

(*a*) Stream processing systems developed by the database community mainly adopt a relational model: events are represented as tuples. All the tuples belonging to the same stream share the same schema, which specifies the number, order, and type of attributes in the tuples.

(*b*) CEP systems developed by the community working on distributed event based systems provide more decoupling among the communication parties and offer less guarantees on the format of information. As in traditional content-based publish/subscribe systems, event notification are encoded as attribute-value pairs.

Only a few proposals targeted the integration of event-driven programming and event composition in programming languages [20, 39, 22]. This approach enables event expressions typechecking at compile time.

Conversely, RLs are often implemented as embedded DSLs. As such, they benefit from the standard typechecking of the host language compiler. Moreover, recent research focused on improving safety guarantees in the execution of reactive programs. Liu and Hudak propose arrows to prevent space leaks in the evaluation of reactive programs [29].

Other solutions adopt advanced type systems. Sculthorpe and Nilsson propose to employ dependent types to ensure that a reactive system is *productive*, i.e., it guarantees to deliver output at all points in time. Krishnaswami *et al.* [27] use linear types to control allocation in a reactive program. By doing this, reactive applications are proved to execute in bounded space without sacrificing higher-order reactive values (i.e., composability of time-changing values).

3.2.7 Interaction with Object Oriented Features

Signals were first defined in Functional Reactive Programming (FRP). Recently, some proposals have been made to introduce and integrate signals in object oriented programming languages [43, 25, 32]. A common solution is to implement signals as object fields.

There exist, however, some open research questions in how to integrate reactive programming in an object oriented environment. A first issue involves the definition of a proper detection and notification model in presence of mutable objects.

Another open problem is the relation between signals and object oriented abstractions such as inheritance an polymorphism. Naively, one can easily envisage that signals can be inherited and referred to from subclasses via *super*. In addition, it is reasonable to define abstract signals, refer to them generically, and late bind them once they are implemented in the subclasses.

Finally, it is an open question to decide whether a signal `s` can be redefined after the first assignment, i.e., whether the expression that defines `s` can change over time. In this context, many researchers agree on forbidding the reassignment of signal fields to reduce complexity [43].

Less attention has been paid in integrating event-based programming, and in particular CEP, in object oriented languages. Nevertheless, some promising proposals exist [20, 39], which also introduce event subtyping and inheritance.

4. RESEARCH AGENDA

Based on the analysis of the synergies and differences of CEP and RLs, we now propose a preliminary research agenda. In particular, we highlight the key aspects that in our opinion could promote further advancements in supporting reactive applications.

Our proposed agenda is shown in Figure 2. It develops along two axes: *integration* of CEP and RLs and *evolution* of the two fields.

4.1 Integration

As our analysis reveals, CEP (and, more in general, event-based programming) and RLs offer complementary solutions to support reactive applications. The similarities of the two approaches appear

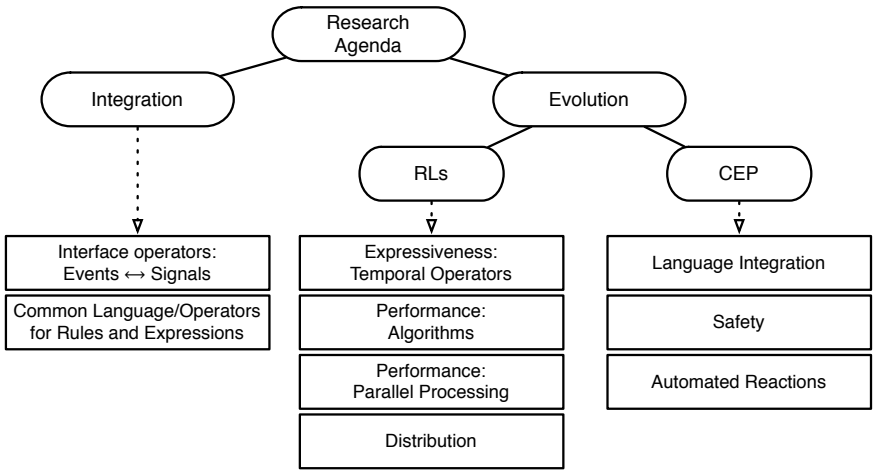

Figure 2: Research Agenda: an Overview

from the high level perspective offered in Figure 1: despite their differences, both CEP and RLs involve the same execution phases to detect, elaborate, and propagate the changes from the sources to the interested components.

For this reason, we believe that a first research direction should target the *integration* of the two solutions, with the attempt to create a coherent environment in which the programming abstractions offered by CEP and RLs could coexist and seamlessly interact.

It is already known from the seminal work on functional reactive programming that signals and events are complementary [17]. A change in the value of a signal can be seen as an (implicit) event occurrence; similarly, signals can change as a reaction to the detection of an event notification. On the other hand, events are traditionally employed in various forms (e.g., Observer pattern, C# events) to structure reactive applications in object oriented languages.

We foresee two possible lines of integration. The first one involves the definition of ad-hoc operators to derive signals from events and vice-versa. A first proposal that goes in this direction is represented by the REScala project [42]. In this work, some built-in operators are introduced to extract events from signals and to define signals from events. One of the key motivations to follow this approach is backward compatibility. Indeed, existing object oriented software mostly implements reactive behaviors using event-based abstractions [42]. The usage of built-in operators that derive signals from events could interface signal-based reactivity to the traditional event-based reactivity.

The second line of integration involves the languages and operators used to define CEP rules and RLs expressions. More in particular, it would be possible to extend RLs expressions to predicate over past values of a signal and consequently to support all the operators traditionally developed for CEP. In this way, RLs can achieve the degree of expressivity and the declarative form of CEP rules even in computations that include time and ordering constraints.

4.2 Evolution

Beside promoting integration, a more detailed analysis of RLs and CEP can drive future *evolutions* in both fields, spreading ideas across heterogeneous communities that tackle similar concerns from different perspectives.

4.2.1 Evolution of RLs

As already observed, RLs currently lack the expressivity of CEP when it comes to deal with time sequences, temporal patterns, and temporal constraints. Extending RLs with a better support for time could constitute a first line of research.

Second, one of the main targets of the community working on CEP is performance. Several data structures, algorithms, and techniques have been proposed to speed up the processing phase and achieve better scalability in the rate of events and in the number of rules. An in-depth analysis of these approaches could help to identify solutions suitable to improve the performance of RLs.

Third, existing RLs apply a sequential computational model. In many cases, however, reactions are computationally independent and parallelism could be used to improve performance (inter-expression parallelism). In other cases, parallel algorithms could be used to speed up the computation of a single, complex, expression (intra-expression parallelism). Both scenarios have been extensively studied in CEP systems and several solutions have been proposed. An interesting research direction could target the implementation of these solutions into RLs.

Finally, RLs have been mostly explored in the local setting. Conversely, CEP systems typically offer support for distribution. Several processing algorithms and communication protocols have been studied for distributed event processing: they can contribute to further advance the research on distributed reactive programming [41]. Starting from the observation that several RLs employ events behind the scenes to propagate changes, the techniques used in CEP for propagating events and distributing them among several hosts can be reused.

4.2.2 Evolution of CEP Systems

CEP and, more in general, event composition functionalities have been usually implemented in dedicated, stand alone components. External clients can typically access these functionalities via local or remote procedure invocation or through message-based network communication.

Despite some promising proposals, finding a good way to integrate CEP abstractions within programming languages vastly remains an open issue. A better analysis of the solutions proposed for RLs could guide future evolutions in this context: programming language-based techniques that have been successfully applied to RLs can be potentially adopted for CEP as well.

Strictly related to the integration within programming languages is the issue of ensuring better safety guarantees. On the one hand, this demands for a more sophisticated data model, which could seamlessly integrate within the type system of programming lan-

guages and enable automated type checking from the compiler. On the other hand, the system could offer a finer-grained control to its users, by enabling them to introduce additional safety contraints, e.g., on resource consumption or on the ordering of event propagations.

Finally, providing automated reactions (e.g., update of values), integrated within CEP rules, could raise the level of abstractions currently offered by CEP systems, offering a more intuitive and concise way to control reactive behaviors.

5. ESSENTIAL RELATED WORK

The fundamental related work has been already introduced throughout the paper. For convenience, a minimal summary of the main fields related to CEP and RLs is outlined in the taxonomy that follows.

Information Flow Processing Systems. While this paper focuses specifically on CEP systems, they belong to a wider class of systems generically defined as Information Flow Processing (IFP) systems [15]. They include all those solutions that are designed to process flows of information on-the-fly, to timely compute or update some results.

Beside CEP, Data Stream Management Systems (DSMSs) [5] are another well known and widely studied class of IFP systems. DSMSs were developed by the database community: they consider streams of tuples and define the computation through *continuous queries*, which isolate portions of the streams using time-based or count-based windows, and apply traditional relational operators on such portions.

CEP systems are often considered as an extension of traditional publish/subscribe middleware systems [21], designed to propagate events based on their type or content. Differently from CEP, the publish/subscribe communication paradigm does not include any operator for event composition.

Event-based languages. Event-based languages support events as dedicated language abstractions. Advanced features include event quantification (Ptolemy [39]), implicit events inspired by aspect-oriented programming (EScala [22]), joins among events (JEScala [46]), and support for event composition, including time windows and joins over events streams (EventJava [20]). Strictly related to event-based languages is the Reactive Extensions (Rx) library [35]. Rx enables developers to represent asynchronous data streams using *observables* and to query them using the LINQ query language. Rx has been implemented in several mainstream programming languages, including Java and C# and is being used in some large-scale projects, including the Netflix streaming media provider.

Functional Reactive Programming. As mentioned in Section 2, the concept of time-changing values (i.e., signals or behaviors) was first introduced in Functional Reactive Programming (FRP). FRP is a programming paradigm proposed in the context of strictly functional languages and originally applied to graphical animations [17].

Synchronous dataflow languages. Synchronous dataflow languages, like Lustre [9] and Esterel [6], define the computation using a reactive network. A signal is synchronously propagated across the network and triggers the computation in the nodes. Synchronous dataflow languages strongly focus on performance and time-bounded execution. For instance, PRET-C [3] offers efficient C-based shared memory communications between concurrent threads. Interestingly, synchronous languages have been proposed as an abstraction for high-level development of Wireless Sensor Network [26].

Self-adjusting computation. Self-adjusting computation is about automatic inference of incremental programs from batch ones [1]. Similarly to reactive languages, the adopted model is a dependency graph where the dependent nodes are updated when a change occurs.

6. CONCLUSION

RLs and CEP systems offer different ways to support reactive applications. RLs automatically update time-changing values, while CEP systems define, detect, and propagate high level composite events starting from the observation of low level primitive events. Despite the similarities between RLs and CEP, research in these fields has been carried on by separate communities.

This paper represents a first step in the direction of analyzing the synergies and differences of the two research areas, which aims at promoting the discussion and the sharing of knowledge between two distant communities.

The result we expect is twofold. First, each community can benefit from results and techniques successfully applied by the other. Second, we hope that such an exchange can encourage research where RLs and CEP integrate and complement each other.

Acknowledgment

This research has been funded by the Dutch national program COMMIT and by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under grant No. 01IC12S01V SINNODIUM.

7. REFERENCES

[1] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, Nov. 2006.

[2] M. Ali. An introduction to Microsoft SQL server Streaminsight. In *Intl. Conference and Exhibition on Computing for Geospatial Research & Application*, page 66. ACM, 2010.

[3] S. Andalam, P. S. Roop, and A. Girault. Deterministic, predictable and light-weight multithreading using pret-c. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 1653–1656. IEEE, 2010.

[4] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, and R. Studer. Etalis: Rule-based reasoning in event processing. In *Reasoning in Event-Based Distributed Systems*, pages 99–124. Springer, 2011.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *ACM PODS*, pages 1–16. ACM, 2002.

[6] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87 – 152, 1992.

[7] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: a high-performance event processing engine. In *Intl. conference on Management of data*, pages 1100–1102. ACM, 2007.

[8] K. Burchett, G. H. Cooper, and S. Krishnamurthi. Lowering: a static optimization technique for transparent functional reactivity. PEPM '07, pages 71–80, New York, NY, USA, 2007. ACM.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. POPL, pages 178–188. ACM, 1987.

[10] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. ESOP'06, pages 294–308. Springer, 2006.

[11] G. Cugola and A. Margara. Raced: an adaptive middleware for complex event detection. In *ACM Intl. Workshop on Adaptive and Reflective MIddleware (ARM)*, page 5. ACM, 2009.

[12] G. Cugola and A. Margara. Tesla: a formally defined event specification language. In *ACM DEBS*, pages 50–61. ACM, 2010.

[13] G. Cugola and A. Margara. Complex event processing with t-rex. *Journal of Systems and Software*, 85(8):1709–1728, 2012.

[14] G. Cugola and A. Margara. Low latency complex event processing on parallel hardware. *Journal of Parallel and Distributed Computing*, 72(2):205–218, 2012.

[15] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[16] G. Cugola and A. Margara. Deployment strategies for distributed complex event processing. *Computing*, 95(2):129–156, 2013.

[17] C. Elliott and P. Hudak. Functional reactive animation. ICFP '97, pages 263–273. ACM, 1997.

[18] Esper performance site. http://docs.codehaus.org/display/ESPER/Esper+performance.

[19] O. Etzion and P. Niblett. *Event processing in action*. Manning Publications Co., 2010.

[20] P. Eugster and K. Jayaram. EventJava: An extension of Java for event correlation. In *ECOOP*, pages 570–594. Springer, 2009.

[21] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[22] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. EScala: modular event-driven object interactions in Scala. AOSD '11, pages 227–240. ACM, 2011.

[23] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. On supporting kleene closure over event streams. In *ICDE*, pages 1391–1393. IEEE, 2008.

[24] P. Hudak. Functional reactive programming. In *Programming Languages and Systems*, pages 1–1. Springer, 1999.

[25] D. Ignatoff, G. H. Cooper, and S. Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *FLOPS*, pages 259–276, 2006.

[26] M. Karpinski and V. Cahill. High-level application development is realistic for wireless sensor networks. In *SECON*, pages 610–619. IEEE, 2007.

[27] N. R. Krishnaswami, N. Benton, and J. Hoffmann. Higher-order functional reactive programming in bounded space. In *POPL*, POPL '12, pages 45–58, New York, NY, USA, 2012. ACM.

[28] G. T. Lakshmanan, Y. Li, and R. Strom. Placement strategies for internet-scale data stream systems. *Internet Computing, IEEE*, 12(6):50–60, 2008.

[29] H. Liu and P. Hudak. Plugging a space leak with an arrow. *Electron. Notes Theor. Comput. Sci.*, 193:29–45, Nov. 2007.

[30] A. Lombide Carreton, S. Mostinckx, T. Cutsem, and W. Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In J. Vitek, editor, *Objects, Models, Components, Patterns*, volume 6141 of *Lecture Notes in Computer Science*, pages 41–60. Springer Berlin Heidelberg, 2010.

[31] D. C. Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

[32] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.

[33] I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. In G. Castagna, editor, *ECOOP'13*, volume 7920 of *LNCS*, pages 707–731. Springer Berlin Heidelberg, 2013.

[34] A. Margara and G. Cugola. High performance publish-subscribe matching using parallel hardware. *IEEE Transactions on Parallel and Distributed Systems*, 2013. To Appear.

[35] E. Meijer. Your mouse is a database. *Commun. ACM*, 55(5):66–73, May 2012.

[36] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. OOPSLA '09, pages 1–20. ACM, 2009.

[37] H. Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. ICFP '05, pages 54–65, New York, NY, USA, 2005. ACM.

[38] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, pages 49–49. IEEE, 2006.

[39] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP*, pages 155–179. Springer, 2008.

[40] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *VLDB*, 3(1-2):1525–1528, 2010.

[41] G. Salvaneschi, J. Drechsler, and M. Mezini. Towards distributed reactive programming. In *Coordination Models and Languages*, pages 226–235. Springer, 2013.

[42] 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 Aspect-Oriented Software Development*, AOSD '14. ACM, 2014. To Appear.

[43] G. Salvaneschi and M. Mezini. Reactive behavior in object-oriented applications: An analysis and a research roadmap. In *AOSD'2013*, 2013.

[44] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *ACM DEBS*, page 4. ACM, 2009.

[45] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *ACM PODS*, pages 250–258. ACM, 2005.

[46] J. M. Van Ham, G. Salvaneschi, M. Mezini, and J. Noyé. JEScala: Modular coordination with declarative events and joins. In *Proceedings of the 13th International Conference on Aspect-Oriented Software Development*, AOSD '14, New York, NY, USA, 2014. ACM. To Appear.

[47] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. ICFP '01, pages 146–156, New York, NY, USA, 2001. ACM.

[48] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. PADL '02, pages 155–172, London, UK, UK, 2002. Springer-Verlag.