A Survey on Reactive Programming

ENGINEER BAINOMUGISHA, ANDONI LOMBIDE CARRETON, TOM VAN CUTSEM, STIJN MOSTINCKX, and WOLFGANG DE MEUTER, Vrije Universiteit Brussel

Reactive programming has recently gained popularity as a paradigm that is well-suited for developing event-driven and interactive applications. It facilitates the development of such applications by providing abstractions to express time-varying values and automatically managing dependencies between such values. A number of approaches have been recently proposed embedded in various languages such as Haskell, Scheme, JavaScript, Java, .NET, etc. This survey describes and provides a taxonomy of existing reactive programming approaches along six axes: representation of time-varying values, evaluation model, lifting operations, multidirectionality, glitch avoidance, and support for distribution. From this taxonomy, we observe that there are still open challenges in the field of reactive programming. For instance, multidirectionality is supported only by a small number of languages, which do not automatically track dependencies between time-varying values. Similarly, glitch avoidance, which is subtle in reactive programs, cannot be ensured in distributed reactive programs using the current techniques.

Categories and Subject Descriptors: D.3.2 [**Language Classifications**]: Dataflow languages; Specialised application languages; D.3.3 [**Language Constructs and Features**]: Concurrent programming structures; Constraints; Control structures; Patterns

General Terms: Design, Languages

Additional Key Words and Phrases: Reactive programming, interactive applications, event-driven applications, dataflow programming, functional reactive programming, reactive systems

ACM Reference Format:
Bainomugisha, E., Lombide Carreton, A., Van Cutsem, T., Mostinckx, S., De Meuter, W. 2013. A Survey on reactive programming. ACM Comput. Surv. 45, 4, Article 52 (August 2013), 34 pages.
DOI: http://dx.doi.org/10.1145/2501654.2501666

1. INTRODUCTION

Today's applications are increasingly becoming highly interactive, driven by all sorts of events originating from within the applications and their outside environment. Such event-driven applications maintain continuous interaction with their environment, processing events and performing corresponding tasks such as updating the application state and displaying data [Pucella 1998]. The most interactive part of such applications is usually the GUI, which typically needs to react to and coordinate multiple events (e.g., mouse clicks, keyboard button presses, multitouch gestures, etc.).

E. Bainomugisha is funded by the SAFE-IS project in the context of the Research Foundation, Flanders (FWO). A. Lombide Carreton is funded by the MobiCrant project in the context of the InnovIris (the Brussels Institute for Research and Innovation).
T. Van Cutsem is a Postdoctoral fellow of the Research Foundation, Flanders (FWO).
Authors' addresses: E. Bainomugisha (corresponding author), A. Lombide Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter, Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Elsene, Brussels, Belgium; email: ebainomu@vub.ac.be.

DOI: http://dx.doi.org/10.1145/2501654.2501666

These applications are difficult to program using conventional sequential programming approaches, because it is impossible to predict or control the order of arrival of external events and as such control jumps around event handlers as the outside environment changes unexpectedly (*inverted control*, i.e., the control flow of a program is driven by external events and not by an order specified by the programmer). Moreover, when there is a state change in one computation or data, the programmer is required to manually update all the others that depend on it. Such manual management of state changes and data dependencies is complex and error prone (e.g., performing state changes at a wrong time or in a wrong order) [Cooper and Krishnamurthi 2006]. Using traditional programming solutions (such as design patterns and event-driven programming), interactive applications are typically constructed around the notion of asynchronous *callbacks* (event handlers). Unfortunately, coordinating callbacks can be a very daunting task even for advanced programmers since numerous isolated code fragments can be manipulating the same data and their order of execution is unpredictable. Also, since callbacks usually do not have a return value, they must perform side-effects in order to affect the application state [Cooper 2008]. As noted by Edwards [2009] and Maier et al. [2010], a recent analysis of Adobe desktop applications revealed that event handling logic contributes to nearly a half of the bugs reported [Järvi et al. 2008]. In the literature, the problem of callback management is infamously known as *Callback Hell* [Edwards 2009]. To reduce the burden faced by programmers, there is a need for dedicated language abstractions to take care of the event handling logic as well as the management of state changes.

The reactive programming paradigm has been recently proposed as a solution that is well-suited for developing event-driven applications. Reactive programming tackles issues posed by event-driven applications by providing abstractions to express programs as reactions to external events and having the language automatically manage the flow of time (by conceptually supporting simultaneity), and data and computation dependencies. This yields the advantage that programmers need not worry about the order of events and computation dependencies. Hence, reactive programming languages abstract over time management, just as garbage collectors abstract over memory management. This property of automatic management of data dependencies can be observed in spreadsheet systems, arguably the most widely used end-user programming language. A spreadsheet typically consists of cells which contain values or formulas. If a value of a cell changes then the formulas are automatically recalculated. Reactive programming is essentially about embedding the spreadsheet-like model in programming languages.

The reactive programming paradigm is based on the synchronous dataflow programming paradigm [Lee and Messerschmitt 1987] but with relaxed real-time constraints. It introduces the notion of *behaviors* for representing continuous time-varying values and *events* for representing discrete values. In addition, it allows the structure of the dataflow to be dynamic (i.e., the structure of the dataflow can change over time at runtime) and support higher-order dataflow (i.e., the reactive primitives are first-class citizens) [Cooper 2008; Sculthorpe 2011]. Most of the research on reactive programming descends from Fran [Elliott and Hudak 1997; Wan and Hudak 2000], a functional domain specific language developed in the late 1990s to ease the construction of graphics and interactive media applications. Several libraries and extensions to various languages have since been conceived to provide support for reactive programming. Other application domains of reactive programming include modeling and simulation [Nilsson et al. 2003], robotics [Peterson and Hager 1999], computer vision [Peterson et al. 2001], and stage lighting [Sperber 2001b].

This article provides a comprehensive survey of the research and recent developments on reactive programming. We describe and provide a taxonomy of existing reactive programming approaches along six axes: representation of time-varying values,

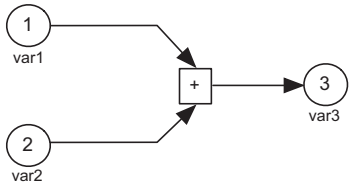

Fig. 1. Graphical representation of expression dependencies in a reactive program.

evaluation model, lifting operations, multidirectionality, glitch avoidance, and support for distribution. We further discuss the techniques and algorithms employed by the existing solutions. From this taxonomy, we identify open issues that still need be tackled in the reactive programming research. In particular, we observe that multidirectionality is only supported by a small number of reactive programming systems that do not automatically track dataflow dependencies. Another open issue is that when applying reactive programming to distributed programming—which is in many cases asynchronous and event driven—glitch avoidance cannot be ensured using the current techniques. With interactive applications (e.g., Web applications) becoming increasingly distributed, we believe that reactive programming needs to be further explored to cover distributed environments while giving the same assurances as their nondistributed counterparts. This article builds on prior surveys [Benveniste et al. 2003; Whiting and Pascoe 1994; Johnston et al. 2004] that review early research on synchronous programming and dataflow programming.

2. REACTIVE PROGRAMMING

Reactive programming is a programming paradigm that is built around the notion of continuous time-varying values and propagation of change. It facilitates the declarative development of event-driven applications by allowing developers to express programs in terms of what to do, and let the language automatically manage when to do it. In this paradigm, state changes are automatically and efficiently propagated across the network of dependent computations by the underlying execution model. Let us explain change propagation with an example.

Consider a simple example of calculating the sum of two variables.

```
var1 = 1
var2 = 2
var3 = var1 + var2
```

In conventional sequential imperative programming, the value of the variable `var3` will always contain 3, which is the sum of the initial values of variables `var1` and `var2` even when `var1` or `var2` is later assigned a new value (unless the programmer explicitly assigns a new value to the variable `var3`). In reactive programming, the value of the variable `var3` is always kept up-to-date. In other words, the value of `var3` is automatically recomputed over time whenever the value of `var1` or `var2` changes. This is the key notion of reactive programming. Values change over time and when they change all dependent computations are automatically reexecuted. In reactive programming terminology, the variable `var3` is said to be dependent on the variables `var1` and `var2`. We depict such a dependency graph in Figure 1.

2.1. Distinguishing Features of Reactive Programming Languages

The two distinguishing features of reactive programming languages are: behaviors and events.

Behaviors. In the reactive programming literature, behaviors is the term used to refer to *time-varying* values. Behaviors continuously change over time and are first-class and composable abstractions [Elliott and Hudak 1997]. A basic example of a behavior is time itself. As such, most reactive programming languages provide a behavior primitive to represent time (e.g., a primitive `seconds` to represent the value of the current seconds of a minute). Other behaviors can easily be expressed as a function of the time primitive provided by the language. For instance, a behavior whose value is `10` times the current seconds can be expressed in terms of `seconds` as `seconds*10`.

Events. Events refer to (potentially infinite) streams of value changes. Unlike behaviors that continuously change over time, events occur at discrete points in time (e.g., keyboard button presses, change in location, etc.). Events and behaviors can be seen as dual to each other and it has been argued that one may be used to represent the other [Cooper 2008]. Like behaviors, events are first-class values and are composable. Most languages provide primitive *combinators* to arbitrarily combine events or filter a sequence of events. For instance, Flapjax [Meyerovich et al. 2009] and FrTime [Cooper and Krishnamurthi 2006] provide `merge` and `filter` combinators.

3. TAXONOMY

This section discusses the six properties that constitute the taxonomy presented in Table I. The taxonomy is formulated along six axes: basic abstractions, evaluation model, lifting, multidirectionality, glitch avoidance, and support for distribution. These properties are discussed in detail next. We give a summary of the taxonomy at the end of this section.

3.1. Basic Abstractions

Just like primitive operators (e.g., an assignment) and values (e.g., a number) are basic abstractions in an imperative language, basic abstractions in a reactive language are reactive primitives that facilitate the writing of reactive programs. Most languages provide behaviors (for representing continuous time-varying values) and events (for representing streams of timed values), which we describe in Section 2.1. These abstractions are usually composable and are used to express a reactive program. Table I gives a summary of the basic abstractions provided by different reactive languages.

Behaviors that represent continuously changing values present a significant implementation challenge. Their continuous nature implies that when the internal clock rate of a reactive program is increased, these behaviors yield more precise values. Depending on the host language, this implementation challenge varies. In Fran and Yampa, the values that continuous behaviors yield can be computed lazily thanks to the lazy nature of their host language Haskell. In nonlazy languages, behaviors appear to the programmer continuously change because they always have a value at any point time, but have to be regularly sampled. This difference allows Fran and Yampa to offer dedicated operations working on continuously changing values, such as `integral` and `derivative`, which are lacking in other languages. They have a strong importance in the domain of modeling and simulation.

3.2. Evaluation Model

The evaluation model of a reactive programming language is concerned with how changes are propagated across a dependency graph of values and computations. From the programmer's point of view, propagation of changes happens automatically. Indeed, this is the essence of reactive programming. A change of a value should be automatically propagated to all dependent computations. When there is an event occurrence at an event source, dependent computations need to be notified about the changes, possibly

Table I. A Taxonomy of Reactive Programming Languages

Language	Basic abstractions	Evaluation model	Lifting	Multidirectionality	Glitch avoidance	Support for distribution
FRP Siblings						
Fran	behaviours and events	Pull	Explicit	N	Y	N
Yampa	signal functions and events	Pull	Explicit	N	Y	N
FrTime	behaviours and events	Push	Implicit	N	Y	N
NewFran	behaviours and events	Push and Pull	Explicit	N	Y	N
Frappé	behaviours and events	Push	Explicit	N	N	N
Scala.React	signals and events	Push	Manual	N	Y	N
Flapjax	behaviours and events	Push	Explicit and implicit	N	Y (local)	Y
AmbientTalk/R	behaviours and events	Push	Implicit	N	Y (local)	Y
Cousins of Reactive Programming						
Cells	rules, cells and observers	Push	Manual	N	Y	N
Lamport Cells	reactors and reporters	Push and Pull	Manual	N	N	Y
SuperGlue	signals, components, and rules	Push	Manual	N	Y	N
Trellis	cells and rules	Push	Manual	N	Y*	N
Radul/Sussman Propagators	propagators and cells	Push	Manual	Y	N	N
Coherence	reactions and actions	Pull	N/A	Y	Y	N
.NET Rx	events	Push	Manual	N	N?	N

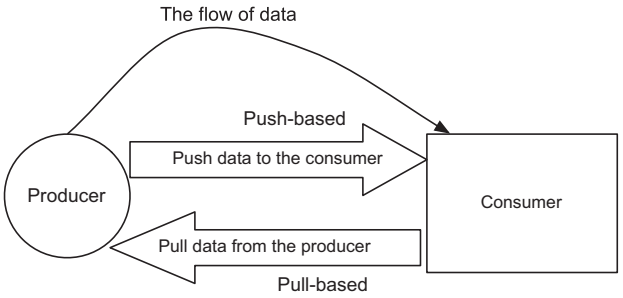

Fig. 2. Push- versus pull-based evaluation model.

triggering a recomputation. At language level, the design decision that needs to be taken into account is who initiates the propagation of changes, that is, whether the source should "push" new data to its dependents (consumers) or the dependents should "pull" data from the event source (producer). In both cases the sequence of values flows from producer to consumer as depicted in Figure 2. In the reactive programming literature, there exist two evaluation models.

3.2.1. Pull-Based. In the pull-based model, the computation that requires a value needs to "pull" it from the source. That is, propagation is driven by the demand of new data (*demand driven*). The first implementations of reactive programming languages such as Fran [Elliott and Hudak 1997] use the pull-based model. The pull model offers the flexibility that the computation requiring the value has the liberty to only pull the new values when it actually needs them. This is thanks to the lazy evaluation of the host language Haskell, where the actual reaction will only happen when it needs to be observed by the outside world.

A major criticism of the pull-based model is that it may result in a significant latency between an event occurrence and when its reaction happens, because all delayed computations must suddenly be performed to lead to the reaction. This can lead to *space* and *time leaks*, which may arise over time. Hudak et al. [2003] describe space and time leaks as "a time leak in a real-time system occurs whenever a time-dependent computation falls behind the current time because its value or effect is not needed yet, but then requires catching up at a later point in time. This catching up can take an arbitrarily long time (time leak) and may or may not consume space as well (space leak)". This issue mainly arises in reactive languages that are implemented in a lazy language such as Fran [Elliott and Hudak 1997] and Yampa [Hudak et al. 2003]. In the recent implementation of Fran, NewFran [Elliott 2009] these issues have been fixed. Yampa [Hudak et al. 2003] avoids these problems by limiting expressiveness through the use of arrows [Hughes 2000] and restricting the behaviors to be non-first class.

3.2.2. Push-Based. In the push-based model, when the source has new data, it pushes the data to its dependent computations. That is, propagation is driven by availability of new data (*data driven*) rather than the demand. This is the approach undertaken by all implementations of reactive programming in eager languages. This usually involves calling a registered callback or a method [Sperber 2001a]. Most recent implementations of reactive programming such as Flapjax [Meyerovich et al. 2009], Scala.React [Maier et al. 2010], and FrTime [Cooper and Krishnamurthi 2006] use a push-based model. Languages implementing the push-based model need an efficient solution to the problem of wasteful recomputations since recomputations take place every time the input sources change. Also, because propagation of changes is data driven, reactions happen as soon as possible [Elliott 2009].

Push versus Pull. Each of the evaluation models has its advantages and disadvantages. For instance, the pull-based model works well in parts of the reactive system where sampling is done on event values that change continuously over time [Sperber 2001a]. Additionally, lazy languages using a pull-based approach yield an advantage with regard to initialization of behaviors. Since their actual values are computed lazily on a by-demand basis, initialization does not have to happen explicitly. Especially continuous behaviors will already yield a value by the time it is needed. In a push-based approach, the programmer must initialize behaviors explicitly to make sure that they hold a value when eagerly evaluating code in which they are used.

A push-based model, on the other hand, fits well in parts of the reactive system that require instantaneous reactions. Some reactive programming languages use either a pull-based or push-based model while others employ both. Another issue with push-based evaluation is glitches, which are discussed in the next section. The approaches that combine the two models reap the benefits of the push-based model (efficiency and low latency) and those of the pull-based model (flexibility of pulling values based on demand). The combination of the two models has been demonstrated in the Lula system [Sperber 2001b] and the most recent implementation of Fran [Elliott 2009].

3.3. Glitch Avoidance

Glitch avoidance is another property that needs to be considered by a reactive language. Glitches are update inconsistencies that may occur during the propagation of changes. When a computation is run before all its dependent expressions are evaluated, it may result in fresh values being combined with stale values, leading to a glitch [Cooper and Krishnamurthi 2006]. This can only happen in languages employing a push-based evaluation model.

Consider an example reactive program.

```
var1 = 1
var2 = var1 * 1
var3 = var1 + var2
```

In this example, the value of the variable `var2` is expected to always be the same as that of `var1`, and that of `var3` to always be twice that of `var1`. Initially when the value of `var1` is `1`, the value of `var2` is `1` and `var3` is `2`. If the value of `var1` changes to, say `2`, the value of `var2` is expected to change to `2` while the value of `var3` is expected to be `4`. However, in a naive reactive implementation, changing the value of `var1` to `2` may cause the expression `var1 + var2` to be recomputed before the expression `var1 * 1`. Thus the value of `var3` will momentarily be `3`, which is incorrect. Eventually, the expression `var1 * 1` will be recomputed to give a new value to `var2` and therefore the value of `var3` will be recomputed again to reflect the correct value `4`. This behavior is depicted in Figure 3.

In the reactive programming literature, such a momentary view of inconsistent data is known as a glitch [Cooper and Krishnamurthi 2006]. Glitches result in incorrect program state and wasteful recomputations and therefore should be avoided by the language. Most reactive programming languages eliminate glitches by arranging expressions in a topologically sorted graph [Cooper and Krishnamurthi 2006; Meyerovich et al. 2009; Maier et al. 2010], thus ensuring that an expression is always evaluated after all its dependents have been evaluated.

Most recent reactive implementations achieve glitch avoidance in reactive programs running on a single computer, but not in distributed reactive programs. Avoiding glitches in a distributed setting is not straightforward because of network failures, delays, and lack of a global clock. This is a potential sweet spot for future research on

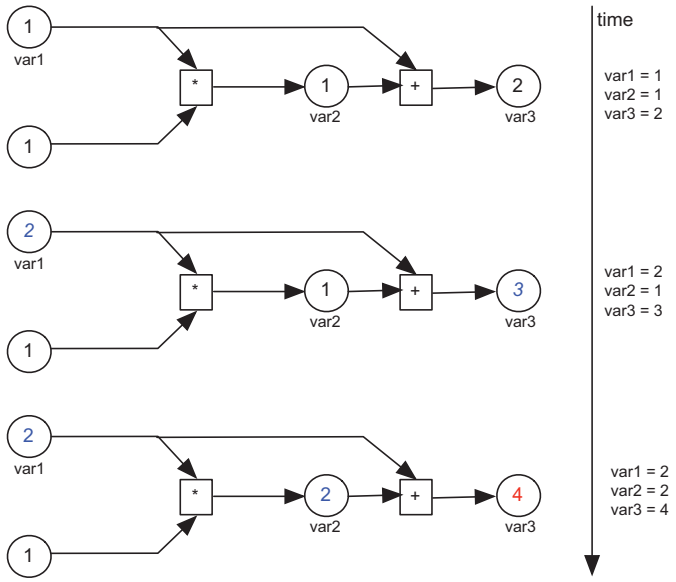

Fig. 3. Glitches: Momentary view of inconsistent program state and recomputation.

distributed reactive systems that provide glitch freedom. We further discuss distributed reactive programming as an open issue in Section 5.

Also, an efficient reactive implementation should avoid unnecessary recomputations of values that do not change. Dependent computations need not be recomputed if the value they depend on is updated to a new value that is the same as the previous value. Taking the same example as earlier, suppose the value of `var1` that is initially `1`, is afterwards updated to the same value (i.e., `1`). In such a case, the values for `var2` and `var3` need not to be recomputed as the value of `var1` remained unchanged.

3.4. Lifting Operations

When reactive programming is embedded in host languages (either as a library or as a language extension), existing language operators (e.g., +, *) and user-defined functions or methods must be converted to operate on behaviors. In the reactive programming literature the conversion of an ordinary operator to a variant that can operate on behaviors is known as *lifting*.

Lifting serves a dual purpose: it transforms a function's type signature (both the types of its arguments and its return type) and it registers a dependency graph in the application's dataflow graph. In the following definitions, we assume functions that take a single behavior argument for the sake of brevity, and generalizing to functions that take multiple arguments is trivial.

$$lift : f(T) \rightarrow f_{lifted}(Behaviour < T >)$$

In this definition, T is a nonbehavior type while $Behavior$ is a behavior type holding values of type T. Therefore, lifting an operator f that was defined to operate on a nonbehavior value transforms it into a lifted version f_{lifted} that can be applied on a behavior.

At time step i, evaluation of a lifted function f called with a behavior yielding values of type T can be defined as follows.

$$f_{lifted}(Behaviour < T >) \rightarrow f(T_i)$$

Here, T_i denotes the value of *Behavior* at time step i.

3.4.1. Relation between Typing and Lifting. In the body of work discussed in this survey, lifting happens in a number of different ways. Before discussing them, it is important to understand the interplay between the semantics of the host language and transforming a function's type signature. In a statically typed language (such as Haskell or Java), a function or method cannot be directly applied onto a behavior. Usually, this means that the programmer must explicitly lift procedures or methods to ensure type safety (or write functions or methods that expect behavior arguments, i.e., their parameters are statically typed *Behavior*). The need for explicit lifting is mitigated in many of the statically typed languages discussed in this survey because these languages offer a rich set of overloaded primitives intended for their particular problem domain that work on behaviors as well. Of course, this is only true when the language is restricted to the domain for which adequate overloaded operators are provided by the language.

In dynamically typed languages, one can pass behaviors as arguments to functions without having to explicitly lift them to satisfy the type system. In these languages, lifting usually happens implicitly by the language. Of course, at one moment primitive operators of the language will have to be applied to arguments of an unexpected type *Behavior* (e.g., computing the sum of two behaviors). In dynamically typed languages, these primitive operators must be properly overloaded to their lifted version (by, for example, using code transformation techniques to generate lifted operators) by the language.

3.4.2. Classification of Lifting Strategies. In this section, we classify the different ways in which the languages that are surveyed in this article support lifting.

Implicit lifting. In the implicit lifting approach, when an ordinary language operator is applied on a behavior, it is automatically "lifted". Implicit lifting makes reactive programming transparent, since programmers can freely use existing operators on behaviors.

$$f(b_1) \rightarrow f_{lifted}(b_1)$$

In this definition, when an ordinary operator f is applied on a behavior b_1, it is implicitly lifted to f_{lifted}. This is the approach undertaken by dynamically typed languages.

Explicit lifting. With explicit lifting, the language provides a set of combinators that can be used to "lift" ordinary operators to operate on behaviors.

$$lift(f)(b_1) \rightarrow f_{lifted}(b_1)$$

In this definition, the ordinary operator f is explicitly lifted using the combinator $lift$ to be able to operate on the behavior b_1. This is the approach that is usually undertaken by statically typed languages. In many cases, reactive programming systems target a particular problem domain, for which it offers a rich set of overloaded primitives that directly work on behaviors. In this survey, we still classify this as explicit lifting, but we mention in the discussion of the language in question when primitive operators are overloaded to deal with behaviors.

Manual lifting. With manual lifting, the language does not provide lifting operators. Instead, the programmer needs to manually obtain the current value of a time-varying value, which can then be used with ordinary language operators.

$$f(b_1) \rightarrow f(currentvalue(b_1))$$

In this definition, the current value of the time-varying value b_1 is obtained that is then used for the ordinary operator f. In languages that do not offer first-class behaviors,

lifting must always happen manually by manually putting values in wrapper values (sometimes called cells) that encode their dataflow dependencies.

Table I shows how reactive languages compare in terms of lifting operations.

3.5. Multidirectionality

Another property of reactive programming languages is whether propagation of changes happens in one direction (unidirectional) or in either direction (multidirectional). With multidirectionality, changes in derived values are propagated back to the values from which they were derived. For example, writing an expression `F = (C * 1.8) + 32`, for converting temperature between Fahrenheit and Celsius, implies that whenever a new value of either F or C is available the other value is updated. This property is similar to the multidirectional constraints in the constraint programming paradigm [Steele 1980]. Table I shows the surveyed reactive languages that provide support for multidirectionality.

3.6. Support for Distribution

This property is concerned with whether a reactive language provides support for writing distributed reactive programs. The support for distribution enables one to create dependencies between computations or data that are distributed across multiple nodes. For example, in an expression `var3 = var1 + var2`, `var1, var2` and `var3` can be located on different nodes. The need for support for distribution in a reactive language is motivated by the fact that interactive applications (e.g., Web applications, mobile applications, etc.) are becoming increasingly distributed. However, there is a catch in adding support for distribution to a reactive language. It is more difficult to ensure consistency in a dependency graph that is distributed across multiple nodes because of distributed programming characteristics (such as latency, network failures, etc.). We further discuss this challenge as of one of the open issues in Section 5. In the Table I, the column for distribution shows how different reactive languages compare in terms of support for distribution.

3.7. Discussion

Table I presents a taxonomy of the reactive programming languages that we discuss in this article. We evaluate the languages along the aforementioned six axes: basic abstractions, evaluation model, lifting, multidirectionality, glitch avoidance, and support for distribution. `Y` in the table under the property column signifies that a language provides the feature while `N` signifies that a language does not provide that feature.

3.7.1. Basic Abstractions. Most reactive languages provide the same basic composable abstractions, behaviors, and events, for representing continuous values and discrete values. In this survey, we consider these languages to be *siblings*, as they inherit their distinctive features from a single progenitor: Fran [Elliott and Hudak 1997]. There are a few reactive languages that do not provide the same primitive abstractions for representation of time-varying values but rather focus on automatic management of state changes. We refer to these languages as *cousins* of reactive programming. Examples of such cousin languages are: Cells [Tilton 2008], Trellis [Eby 2008], .NET Rx [Hamilton and Dyer 2010], and Radul/Sussman propagators [Radul and Sussman 2009]. Instead, the basic abstractions provided by these languages are for creating dependencies between value producers and consumers.

In short, the most prominent difference between the sibling and cousin languages is that the sibling languages provide abstractions to represent time-varying values, which interoperate with the host language through implicit or explicit lifting. Cousin languages, on the other hand, offer “containers” or “cells” of which the value can vary

over time. The programmer must manually store values in these cells and manually extract them again. In response, the system takes care of correctly propagating changes throughout the network of cells[1]. In practice this means that in sibling languages time-varying values can be used in ordinary expressions (albeit with explicitly lifted operators in some cases) in which dataflow dependencies are automatically tracked, while "cousin" languages require dedicated code to encode these dependencies.

3.7.2. Evaluation Model. Besides NewFran [Elliott 2009] and Lamport Cells [Miller 2003] which employ both the push- and pull-based evaluation models, most reactive languages are either purely pull or pushbased. For efficiency reasons (instantaneous reactions), most recent implementations are purely pushbased.

3.7.3. Lifting. In languages that require the explicit tracking of dataflow-dependent values by the programmer instead of relying on behaviors, lifting must always happen manually by putting values in wrapper values (sometimes called cells) that encode their dataflow dependencies. This is the case for Radul/Sussman propagators, Cells, Lamport Cells, Superglue, Scala.React, Trellis and .NET Rx. They do not provide any lifting operations, which implies that the programmer has to manually retrieve a value of a behavior in order to use it with primitive operations. Languages that track dataflow dependencies between expressions (and hence automatically for values) fall into two categories. As mentioned earlier in Section 3.4.1, this is to be attributed to the type system of their host language. A number of statically typed languages, which include Fran, Yampa, NewFran, and Frappé, require explicit lifting. However, some of them are heavily targeted towards a particular problem domain (e.g., animation for Fran and NewFran), for which they offer an extensive set of lifted operators. This is possible for host languages that support operator overloading, such as Haskell.

Dynamically typed languages such as FrTime (Scheme) and AmbientTalk/R offer implicit lifting. For Flapjax, this is only true if the Flapjax to JavaScript compiler is used. It applies the necessary code transformations to generate lifted operators. When used as a library, lifting must happen explicitly. FrTime uses Scheme's macro system for this and AmbientTalk/R its reflection and metaprogramming facilities. Coherence [Edwards 2009] does not require any lifting operations since the default semantics of the language is reactivity.

3.7.4. Multidirectionality. Multidirectional propagation of changes is a feature that is only supported by Coherence [Edwards 2009] and Radul/Sussman propagators [Radul and Sussman 2009]. The other reactive languages limit propagation of changes to happen in one direction. We further discuss the support for multidirectionality in reactive programming languages as an open issue in Section 5.

3.7.5. Glitch Avoidance. Most of the surveyed reactive languages achieve glitch freedom. However, there are a few exceptions. Radul/Sussman propagators are not implemented with support for glitch avoidance. Programmers need to employ other techniques such as dependency-directed tracking [Stallman and Sussman 1977; Zabih et al. 1987] that can be easily expressed in the base propagator infrastructure. Trellis has some support for glitch avoidance, however, programmers have to take extra care on program structuring in order to realize glitch freedom (thus indicated as `Y*`). .NET Rx is an ongoing project and there is no clear description of the semantics of how glitches are avoided (thus indicated as `N?` in the table).

[1]Coherence [Edwards 2009] is an exception to this classification: it does not provide the classical abstractions (behaviors and event sources), but still tracks dependencies implicitly. Coherence differs from the other languages in the fact that its underlying evaluation model is not dataflow, but a different flavor called *coherent reaction*.

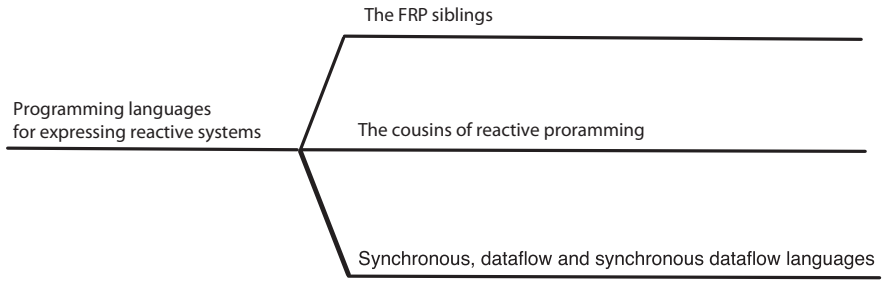

Fig. 4. Classification of Languages for Reactive Programming.

3.7.6. Support for Distribution. Distribution is a feature that is not well-supported in reactive programming languages. We found only three languages: Flapjax [Meyerovich et al. 2009], Lamport Cells [Miller 2003], and AmbientTalk/R [Carreton et al. 2010] that provide support for writing distributed reactive programs. However, even in these languages glitches are avoided only in a local setting (thus indicated `Y(local)`). This observation shows that there is still need for research in the field of distributed reactive programming.

In Section 4 we give a full review for each language listed in Table I.

4. LANGUAGE SURVEY

Our survey consists of 15 representative reactive programming languages that we selected based on the availability of their publications. Most of the reactive languages surveyed are built as language extensions to existing languages. Several reactive languages are extensions to (mostly) functional programming languages such as Haskell and Scheme.

For this survey we sort the reactive languages into three categories: the Functional Reactive Programming (FRP) siblings, the cousins of reactive programming, and synchronous, dataflow, and synchronous dataflow languages. The classification is depicted in Figure 4. We discuss each language based on the features outlined in Section 3. We illustrate each language with a simple temperature converter example. Where necessary we further illustrate the language with an example of drawing a circle on the screen that changes color when the left or right mouse button is pressed.

4.1. The FRP Siblings

The reactive programming languages in this category provide composable abstractions (behaviors and events). In addition, they typically provide primitive combinators for composing events and *switching combinators* to support the dynamic reconfiguration of the dataflow and support higher-order dataflow.

One important point is that the literature on these languages sometimes uses slightly different terminology for behaviors and events, while sometimes the abstractions truly differ. In the following, we will always relate the offered abstractions to our terminology of behaviors and events and stick to this where possible.

In FRP languages, the arguments to functions vary over time and automatically trigger propagation whenever values change. That is, the function is automatically reapplied as soon as one of the arguments changes. FRP allows programmers to express reactive programs in a declarative style. For instance, a sample functional program to draw a circle on the screen at the current mouse position can be easily expressed as `(draw-circle mouse-x mouse-y)`. In this expression, whenever the value of `mouse-x` or `mouse-y` changes, the function `draw-circle` is automatically reapplied to update

Table II. Functional Reactive Programming (FRP) Siblings

Language	Host language
Fran [Elliott and Hudak 1997]	Haskell
Yampa [Hudak et al. 2003]	Haskell
Frappé [Courtney 2001]	Java
FrTime [Cooper and Krishnamurthi 2006]	PLT Scheme (now known as Racket)
NewFran [Elliott 2009]	Haskell
Flapjax [Meyerovich et al. 2009]	JavaScript
Scala.React [Maier et al. 2010]	Scala
AmbientTalk/R [Carreton et al. 2010]	AmbientTalk

the circle position. Elliott and Hudak [1997] identify the key advantages of the FRP paradigm as: *clarity, ease of construction, composability*, and *clean semantics*.

FRP was introduced in Fran [Elliott and Hudak 1997], a language specially designed for developing interactive graphics and animations in Haskell. Since then, FRP ideas have been explored in different languages including Yampa, FrTime, Flapjax, etc. Innovative research on reactive programming has been mostly carried out in the context of FRP. It is therefore not surprising that a large number of the surveyed languages evolve around the notion of FRP. We review eight languages (outlined in Table II) in this category.

Fran

Fran (Functional Reactive Animation) [Elliott and Hudak 1997] is one of the first languages designed to ease the construction of interactive multimedia animations. Fran was conceived as a reactive programming library embedded in Haskell. Its main goal is to enable programming interactive animations with high-level abstractions that provide ways of expressing what the application does and let the language take care of *how* the interaction occurs.

Fran represents continuous time-varying values as behaviors while discrete values are represented as events. Both behaviors and events are first-class values and can be composed using combinators. Behaviors are expressed as reactions to events or other behaviours. In other words, behaviors are built up from events and other behaviurs using combinators.

As Haskell is a statically typed host language, Fran provides lifting operators that transform ordinary Haskell functions into behaviors. Hence, lifting must happen explicitly using these operators. In addition, Fran offers a rich set of overloaded primitive operators that are lifted to work on behaviors as well. Many of these primitives are specifically targeted to its problem domain: animation. In many cases, they eliminate the need to explicitly lift operators for this particular problem domain.

The first implementation of Fran employs a purely pull-based evaluation model. However, the recent implementation of Fran [Elliott 2009] (that we refer to as *NewFran* in this article) combines push- and pull-based evaluation models. The combination of these models yields the benefit of values being recomputed only when they are necessary, and almost instantaneous reactions. The temperature conversion example can be realized in Fran as follows.

```
tempConverter :: Behavior Double
tempConverter = tempF
    where
      tempC = temp
      tempF = (tempC*1.8)+32
```

`tempConverter` is a function that returns a behavior whose value at any given point in time is the value of the current temperature in degrees Fahrenheit. We assume that there is a predefined behavior `temp` whose value at any given time is the current temperature in degrees Celsius.

In order to illustrate Fran's support for the dynamic dataflow structure and high-order reactivity, we consider an example of drawing a circle on the screen and painting it red. The color of the circle then changes to either green when the left mouse button is pressed, or red when the right mouse button is pressed. This example also appears in Elliott and Hudak [1997]. Such a program in Fran can be easily expressed as follows.

```
drawcircle :: ImageB
drawcircle = withColor color circle
    where
     color = stepper red (lbp -=> green .|. rbp -=> red)
```

In the preceding example, `circle` is a predefined behavior for a circle while `lbp` and `rbp` are events that represent left button presses, and right button presses, respectively. The merge operator `.|.` produces events when either input events have an occurrence. We use the `stepper` combinator to create the `color` behavior that starts with `red` until the first button press at which point it changes to either `green` or `red`. `withColor` is a predefined function that takes as argument the `color` behavior and paints the circle with the color. Since `color` is a behavior, the function `withColor` will be automatically reapplied when the `color` behavior gets a new value.

Yampa

Developed at Yale University, Yampa [Hudak et al. 2003] is a functional reactive language that is based on Fran. Like Fran, Yampa is embedded in Haskell. It is specially designed for programming reactive systems where performance is critical. In Yampa, a reactive program is expressed using *arrows* (a generalization of monads) [Hughes 2000] which reduce the chance of introducing the problems of space and time leaks.

The basic reactive abstractions in Yampa are signal functions and events. Signal functions differ from behaviors in the sense that they are functions that encapsulate time-varying values, but are similar in the sense that they are first class. Events in Yampa are represented as signal functions that represent an event stream (event source) that yields an event carrying a certain value at any point in time. Yampa provides primitive combinators for composing events (e.g., merging events). Additionally, it provides a set of switching combinators to provide support for the dynamic dataflow structure. Its *parallel* switching combinators allow the support for dynamic collections of signal functions that are connected in parallel. Signal functions can be added or removed from such a collection at runtime in reaction to events.

One of the main differences between Yampa and Fran is that Fran assumes an implicit, fixed "system input" (e.g., the animation loop), which through which input sources like the mouse or keyboard connect to the program. In Yampa, such input sources are explicit, which is why signal functions have an input and output type, while behaviors in Fran just have an output type.

Yampa provides lifting operators to explicitly lift ordinary functions to the level of signal functions. As in Fran, Yampa also provides a set of overloaded lifted operators. Like the earlier version of Fran, the evaluation model of Yampa is pull based and employs the same techniques as in Fran to avoid glitches. Multidirectional propagation

of changes is not supported. The temperature conversion example can be realized in Yampa as follows.

```
tempConverter = proc -> do
    tempC <- tempSF
    tempF <- (tempC*1.8)+32
returnA -< tempF
```

The previous code snippet shows a reactive program in Yampa using the arrow syntax. `tempConverter` is a signal function that is defined using the `proc` keyword. `proc` is similar to the λ in λ-expressions except that it defines a signal function instead of a procedure. We assume that there is a predefined signal function `tempSF` whose value is the current temperature in degrees Celsius. The conversion of `tempC` to degrees Fahrenheit is bound to the variable `tempF` that is returned as the value whenever the signal function is accessed. The reactive machinery is taken care of by the underlying arrows. The arrow notation avoids the need for explicit lifting if only the instantaneous values need to be observed (i.e., without tracking and propagating changes over time).

To further illustrate Yampa's events and signal function operators, we show the implementation of the example of drawing a circle on the screen that starts with color red and then changes to either green when the left mouse button is pressed or red when the right mouse button is pressed. This example can be expressed in Yampa as follows.

```
drawCircle = proc input -> do
    lbpE      <- lbp -< input
    rbpE      <- rbp -< input
    redB      <- constantB red
    thecolor  <- selectcolor (lbpE 'lmerge' rbpE)
    color     <- rSwitch (redB thecolor)
returnA -< circle 0 0 1 1 color
```

In the preceding example, we extract `lbpE` and `rbpE` from the user input `input` using the `lbp` and `rbp` signal functions. We then use the `lmerge` combinator to combine the two events into a single event that is then used to determine color using the `selectcolor` function. The `lmerge` works like the merge operator `.|.` in Fran but gives the precedence to the left event in case the two events occur at the same time. We use the `rSwitch` combinator (which is similar to the `stepper` in Fran) to create the `color` behavior which starts with `redB` until the first button press at which point it changes to value of `thecolor`. The `circle` function will be automatically reapplied whenever `color` gets a new value.

FrTime

Developed at Brown University, FrTime [Cooper and Krishnamurthi 2006] is a functional reactive programming language extension to Scheme that is designed to ease the development of interactive applications. It runs in the DrScheme (which is now known as Racket) environment [Felleisen et al. 1998] and allows programmers to seamlessly mix FrTime code and pure Scheme, thus enabling reusability of existing libraries such as the GUI toolkits. FrTime also supports a *Read-Eval-Print Loop* (REPL) that enables interactive development and allows users to submit new program fragments dynamically. It achieves this by allowing the reactive engine and the REPL to run concurrently in different threads [Cooper 2008].

The basic reactive abstractions in the language are *behaviors* and *event streams* (which are just events in our terminology) to represent continuous time-varying values

and discrete values, respectively. FrTime provides *hold* and *changes* operations for converting behaviors to events, and vice versa. The *hold* primitive is similar to the `stepper` in Fran. It consumes an initial value and returns a behavior with the initial value as its start value and changes to the event value whenever there is a new event occurrence. On the other hand, the *changes* primitive consumes a behavior and produces an event value every time the behavior changes.

Since Scheme is a dynamically typed language, when primitive Scheme functions are applied to FrTime behaviors, they are automatically lifted (implicit lifting) to behaviors. Ordinary Scheme functions cannot be applied to events. Instead, the language provides a set of event processing combinators (e.g., the *filter* and *map* combinators) that can be applied to events in order to obtain new events.

FrTime's evaluation model is purely push based. A FrTime program is represented as a graph of dataflow dependencies with the nodes corresponding to program expressions while the edges correspond to the flow of values between expressions. As the evaluation model is push based, the propagation of changes across the graph is initiated by the event sources (e.g., a mouse button click). Whenever a new value arrives at an event source, all the computations that depend on it are scheduled for reexecution. To avoid wasteful recomputations, the language makes sure that the computations dependent on values that did not change are not scheduled for execution.

The language avoids glitches by executing dependent computations in a topologically sorted order. Each node is assigned a height that is higher than that of any nodes on which it depends. The nodes are then processed in a priority queue using the heights as the priority. The dataflow graph must be acyclic in order to avoid nonterminating propagation of changes. This glitch avoidance technique has been adopted by other reactive languages such as Scala.React [Maier et al. 2010] and Flapjax [Meyerovich et al. 2009], among others. The temperature conversion example can be realized in FrTime as follows.

```
(define (temp-converter)
  (let* ((tempC temperature)
         (tempF (+ (* tempC 1.8) 32)))
    tempF))
```

The `temp-converter` function returns a behavior whose value at any point in time is the current temperature in degrees Fahrenheit. We assume that there is a predefined behavior `temperature`. FrTime supports implicit lifting, that is, the operators `+` and `*` are implicitly lifted to operate on behaviors.

We further illustrate FrTime by implementing the example of drawing a circle that changes color according to the left or right mouse button press.

```
(define (drawcircle)
  (let ((radius 60)
        (color (new-cell "red")))
    (map-e (lambda (e) (set-cell! color "green")) left-clicks)
    (map-e (lambda (e) (set-cell! color "red")) right-clicks)
    (display-shapes
     (list
      (make-circle mouse-pos radius color)))))
```

In the preceding example, the `make-circle` function constructs a red circle of radius 60 at the current mouse position and changes to green or red when the left or right mouse button is pressed. `mouse-pos` is a predefined behavior whose value is the current mouse position. `color` is a behavior whose initial value is `"red"` (created using the `new-cell` construct). We use the FrTime combinator `map-e` to transform `left-clicks`

and `right-clicks` events and derive values for the `color` behavior. When either the `color` behavior or the `mouse-pos` behavior gets a new value, the `make-circle` function is automatically reapplied to its arguments.

Flapjax

Flapjax [Meyerovich et al. 2009] is a reactive programming language for Web programming that is embedded in JavaScript. The design of Flapjax is mostly based on FrTime [Cooper and Krishnamurthi 2006]. Flapjax can be used as either a JavaScript library or as a language that is compiled to JavaScript. Flapjax introduces two data abstractions to JavaScript: an event stream which represents a stream of discrete events (just events in our terminology) and a behavior which represents a continuous time-varying value whose changes propagate automatically to all dependent values.

Flapjax supports both explicit and implicit lifting of JavaScript functions into behaviors. When used as a library, the programmer needs to explicitly use the lifting function `liftB`. When used as a language, the Flapjax compiler automatically transforms regular JavaScript function invocations into invocations of the explicit lifting function. The compiler also enables Flapjax code to interoperate with JavaScript code. It is possible to call JavaScript functions from Flapjax and vice versa.

Like in FrTime, Flapjax's evaluation model is push based. Flapjax constructs a dataflow graph from events to sinks and whenever there is an event occurrence, its value is pushed through the graph. Nodes of the graph represent computations that are run when an event is received which in turn may propagate to other dependent nodes. As in FrTime [Cooper and Krishnamurthi 2006], glitches are avoided by processing the dependency graph in a topological order.

In Flapjax, developers can write distributed reactive programs on top of the AJAX library since it allows interoperation of JavaScript and Flapjax. For this, it provides built-in event abstractions that represent asynchronous client/server requests and responses as events, which can be processed using the standard event operators/combinators, and can be easily converted to behaviors. This way, client/server interactions become a source of change, just like user input. The reactive programming paradigm allows to concisely combine both in the application logic. However, glitches are not avoided in the resulting distributed reactive applications (as acknowledged by the authors). Since interactive Web applications are a prominent problem domain, we discuss support for distribution as an open issue Section 5.

In Flapjax the temperature conversion example can be expressed as follows.

```
function tempConverter() {
  var temp = Temperature();
  var tempC = temp;
  var tempF = tempC * 1.8 + 32;
  insertValueB(tempC, "tempCtext", "innerHTML");
  insertValueB(tempF, "tempFtext", "innerHTML");
}

<body onLoad = "tempConverter()">
<div id= "tempCtext"> </div>
<div id= "tempFtext"> </div>
</body>
```

The `tempConverter` function implements the functionality of converting temperature from degrees Celsius to degrees Fahrenheit. We assume that there is a predefined behavior `Temperature` whose value at any given point in time is the current temperature.

The `insertValueB` function inserts the values of the behaviors `tempC` and `tempF` in the DOM elements.

We further illustrate Flapjax's support for first-class behaviors and primitive combinators using the example of drawing on a circle a screen that changes color depending on whether a left or right mouse button is pressed. The example can be expressed as follows.

```
//draw circle at (x,y) and paint it color
function drawcircle(x, y, color) {...};

//map button press to color
function handleMouseEvent(evt) {...};

var buttonE = extractEventE(document,"mousedown");
var colorE = buttonE.mapE(handleMouseEvent);
var colorB = startsWith(colorE, "red");
var canvas = document.getElementById('draw');
drawcircle(mouseLeftB(canvas), mouseTopB(canvas), colorB);
```

In the previous example, we use Flapjax's combinators `extractEventE` and `mapE` to extract `mousedown` events from the DOM and transform them into color events. The function `handleMouseEvent` defines the logic of transforming button presses to color. The `startsWith` combinator is similar to the `stepper` in Fran. It takes as arguments the color event `colorE` and initial value `"red"` creates a behavior with the initial value as the red color and changes value to green or red whenever a mouse button press event occurs. The `drawcircle` function takes as argument the mouse position and color behaviors and draws the circle on the screen when the mouse position or color changes. `mouseLeftB` and `mouseTopB` are Flapjax's combinators that create a behavior carrying the x- or y-coordinate of the mouse, relative to the specified DOM element.

Frappé

Frappé [Courtney 2001] is a functional reactive programming library for Java. It extends the JavaBeans component model [Oracle 1997] with a set of classes that correspond to functional reactive programming combinators. In Frappé, a reactive program is constructed by instantiating JavaBeans classes and connecting the components using the FRP combinators. Frappé defines two Java interfaces, `FRPEventSource` and `Behavior`, which provide methods for the basic abstractions of behaviors and events. Concrete classes providing the events functionality (such as raising an event) must implement the `FRPEventSource` interface. Similarly, concrete classes providing the behaviors functionality (such as creating a behavior and behavior combinators) must implement the `Behavior` interface.

Since Java is a statically typed language, lifting of regular Java methods to behaviors is accomplished explicitly by calling the method `liftMethod`. JavaBeans properties may be converted to FRP behaviors using the `makeBehavior` method. Similarly, Frappé provides a method `makeFRPEvent` to convert JavaBeans events into FRP events. It is also possible to use JavaBeans properties as output sinks for Frappé behaviors. Such JavaBeans properties need to be mutable.

Propagation of change in values in Frappé is push based[2]. Whenever there is a value change (e.g., an event occurrence) the Java runtime invokes the appropriate event

[2]Although it has been stated in Cooper [2008] and Maier et al. [2010] that the evaluation model of Frappé mixes push- and pull-based models, our experimentation with Frappé reveal that Frappé employs the push-based model and glitches may occur.

handler on the Frappé object implementing the behavior or event primitive. The primitive event handler in turn invokes the event handler of each registered listener. This is achieved by calling the `eventOccurred` method for events or the `propertyChanged` method for the behavior change. In Frappé, glitch avoidance is not ensured. The temperature conversion example can be expressed in Frappé as follows.

```
Temperature temp = new Temperature();
Behavior tempC = FRPUtilities.makeBehavior(sched, temp,
                                           "currentTemp");
Behavior tempF = FRPUtilities.liftMethod(sched, temp,
           "temperatureConverter", new Behavior[]{tempC});
```

This is assuming that there is a JavaBean `Temperature` that provides a bound property `currentTemp` whose value at any point in time is the current temperature in degrees Celsius. It also provides a `temperatureConverter` method for converting the temperature to degrees Fahrenheit. A behavior is created from the `Temperature` Bean using the `FRPUtilities.makeBehavior` method. The argument `sched` is a global scheduling context used by the Frappé implementation. The method `FRPUtilities.liftMethod` is used to lift `temperatureConverter` to work on the behavior `tempC`. It returns the behavior that is bound to the variable `tempF` whose value at any point in time is the current temperature in degrees Fahrenheit.

We further illustrate Frappé's support for reactive programming by implementing the example of a circle that changes color depending which mouse button is pressed.

```
Drawable circle    = new ShapeDrawable(
                                new Ellipse2D.Double(-1,-1,2,2));
FRPEventSource lbp = FRPUtilities.makeFRPEvent(sched,
                                frame, "franMouse","lbp");
FRPEventSource rbp = FRPUtilities.makeFRPEvent(sched,
                                frame, "franMouse","rbp");

FRPEventSource lbpgreen = new EventBind(sched, lbp,
              FRPUtilities.makeComputation(new ConstB(Color.green)));
FRPEventSource rbpred = new EventBind(sched, rbp,
              FRPUtilities.makeComputation(new ConstB(Color.red)));
FRPEventSource colorE = new EventMerge(sched,
                                       lbpgreen, rbpred);

Behavior colorB = new Switcher(sched,
                          new ConstB(Color.red), colorE);
Behavior anim = FRPUtilities.liftMethod(sched,
                new ConstB(circle), "withColor",
                new Behavior[] {colorB});
```

In the preceding example, we use the `ShapeDrawable` class to create the circle. `lbp` and `rbp` are button press events that are created from JavaBeans using the `FRPUtilities.makeFRPEvent` method. The button press event occurrences are then bound to the color values `green` and `red` using the `EventBind` combinator. We use the `EventMerge` combinator to create the `colorE` events that carry the `lbpgreen` or `rbpred` occurrence. The `Switcher` switching combinator is similar to the `stepper` in Fran. It creates the `colorB` behavior whose initial value is color red. `colorB` changes to red or

green when the left or right button is pressed. Whenever the color changes the circle is redrawn using the `withColor` method.

AmbientTalk/R

AmbientTalk/R [Carreton et al. 2010] is a reactive extension to AmbientTalk [Cutsem et al. 2007], which is an actor-based language that is specially designed for developing mobile applications. AmbientTalk/R integrates reactive programming with the imperative prototype-based object model of AmbientTalk. The basic reactive abstractions in AmbientTalk/R are based on those found in Fran [Elliott and Hudak 1997] and FrTime [Cooper and Krishnamurthi 2006]. AmbientTalk/R provides events and behaviors. Like Flapjax [Meyerovich et al. 2009], AmbientTalk/R can be used as either a library or as part of the language.

Events in AmbientTalk/R are realized as first-class messages that are emitted at discrete points in time. Behaviors in AmbientTalk/R are used to represent time-varying values. AmbientTalk/R provides a `snapshot` operation that allows capturing of a "snapshot" of a behavior at a certain point in time. However, unlike behaviors where a change in value is automatically propagated to all its dependents, a snapshot value does not trigger change propagation. In AmbientTalk/R, behaviors are derived from events. For example, the mouse position can be derived from the `mouseEvent` events provided by the language.

Ordinary AmbientTalk operations are implicitly lifted to behaviors. When a behavior is passed as argument to an AmbientTalk function or method, the result of that invocation is itself a behavior. However, there are primitives whose semantics are preserved and are not automatically lifted to behaviors. For example, the `snapshot` operator always returns a plain value.

In AmbientTalk/R, the evaluation strategy is push based (i.e., events trigger computation). The glitch avoidance technique employed by AmbientTalk/R is similar to that in FrTime [Cooper and Krishnamurthi 2006], Flapjax [Meyerovich et al. 2009], and Scala.React [Maier et al. 2010]. The language maintains a topologically sorted dependency graph that is sorted based on the heights of the dependencies.

AmbientTalk/R builds on the actor-based distributed programming model of AmbientTalk to provide support for distributed reactive programming. Reactive values need not reside on a single host and can be distributed onto multiple hosts. However, unlike in the local setting where there is glitch avoidance assurance, it is not the case in a distributed setting. We further discuss distributed reactive programming as an open issue in Section 5.

The following example shows how to realize the temperature conversion example in AmbientTalk/R.

```
def temperatureConverter := object: {
    def @Reactive temp := Temperature.new();
    def tempC := temp;
    def tempF := tempC * 1.8 + 32;
}
```

In this example, the `object:` keyword creates a fresh object that is bound to the variable `temperatureConverter`. The object includes a reactive field `temp` which is defined using the annotation `Reactive` and is initialized to the current temperature value. The variable `tempC` has a dependency on the reactive variable `temp`. Similarly, the variable `tempF` creates a dependency on the variable `tempC`. Therefore, whenever the value of `temp` changes, the value of `tempC` is updated and the value of `tempF` is recalculated.

The example of a circle that changes color depending which mouse button is pressed can be implemented in AmbientTalk/R as follows.

```
// draw a circle object
def drawCircle(circle) { ... };

def @Reactive circle := object: {
  def posx := 0;
  def posy := 0;
  def color := Color.red;
};

def circleEventSource := changes: circle;
circleEventSource.foreach: { |circle| drawCircle(circle);

def handleMouseClickEvent(e) {
  // Update the circle object's coordinates and color given e.
};

canvas.addMouseClickListener(handleMouseClickEvent);
```

The `circle` object is a behavior which can be updated by setting its fields. This happens in the `handleMouseClickEvent` procedure which is registered as a listener to detect mouse clicks. Before that, an event source is extracted from the `circle` behavior to draw a circle with the correct color for each mouse click.

Scala.React

Scala.React [Maier et al. 2010] is an extension of Scala [Odersky et al. 2004] with the goal of providing reactive programming abstractions in place of the observer pattern. Scala.React provides a general interface that represents generic events. The general interface is represented as a type parameterized Scala class that provides methods to create events and raise events. Reacting to events involves registering a closure on a particular event source. Scala.React also provides operations for composing multiple events into one.

Behaviors in Scala.React are known as *signals*. A signal expression continuously evaluates to a new signal value and automatically takes care of synchronization of data changes and dependencies. Signals are used to create dependencies among variables.

In Scala.React, a signal `c` that depends on the sum of two signals `a` and `b` must be created as follows.

```
val c = Signal{ a()+b() }
```

First, the current values of `a` and `b` must be extracted by calling the closure that encapsulates their current value before they can be summed into a new signal `c`. Although this is manual lifting, we still classify Scala.React as a "sibling" language because of the automatic tracking of dependencies.

Scala.React employs a push-based model for the propagation of changes. As in FrTime [Cooper and Krishnamurthi 2006] and Flapjax [Meyerovich et al. 2009], glitches are avoided by processing the dependency graph in a topological order. It deals with dynamic dependencies by aborting the current evaluation in case its level is found to be higher than the previous one. Then the affected signal is assigned a higher level and rescheduled for validation in the same propagation cycle.

The temperature conversion example can be realized in Scala.React as follows.

```
val tempC = Signal{ Temperature() }
val tempF = Signal{ tempC() * 1.8 + 32}

observe(tempC) { C =>
  // print on label
}
observe(tempF) { F =>
  // print on label
}
```

This is assuming that there is a predefined signal `Temperature` whose value at any given point in time is the current temperature in degrees Celsius. The preceding code snippet uses the `Signal` function to create signals `tempC` and `tempF`. The `tempC` is dependent on the `Temperature` signal while `tempF` is dependent on the `tempC`. In Scala.React, signals are referred to through a function call in the form of `signalName()`. The `observe` method accepts a closure that is executed whenever the signal value changes. In the previous example, the closures are used to display the values of `tempC` and `tempF` in the GUI.

In Scala.React, the example of a circle that changes color depending on the left or right button press can be expressed as follows.

```
val selectedcolor = mouseDown map {md =>
//transform button press events to color signal
}
val color = selectedcolor switchTo Signal{Color.red}
observe(color) { c =>
 // redraw circle
}
```

The preceding example uses the `map` combinator to extract the kind of button press from the `mouseDown` events. The resulting event value is used to create the `selectedcolor` signal whose value corresponds to the left or right button press. Then, we create the `color` signal from the `selectedcolor` or the `SignalColor.red` signal. Initially, `color` holds the current value of the `SignalColor.red` and then switches to that of the `selectedcolor` signal when the left or right button press event occurs. Every time the `color` signal gets a new value, the closure of the `observe` method is automatically invoked resulting in the circle to be redrawn.

4.2. The Cousins of Reactive Programming

As discussed in Section 3.7, there are a few reactive languages that do not provide primitive abstractions for representation of time-varying values and primitive switching combinators for dynamic reconfiguration but provide support for automatic propagation of state changes and other features of reactive programming such as glitch avoidance. Since their abstractions for representing time-varying values do not integrate with the rest of the language, lifting must always performed manually by the programmer in these languages. We refer to these languages as cousins of reactive programming. Table III outlines the reactive languages in this category.

As with the FRP siblings, we illustrate each cousin reactive language with the temperature conversion example. However, since these languages do not provide event and behavior combinators such as `merge`, `map-e`, and `switch`, it is difficult to express some applications such as that of drawing a circle that starts with a color red and switches to green or red depending on whether the left or right mouse button is pressed.

Table III. The Cousins of Reactive Programming

Language	Host language
Cells [Tilton 2008]	CLOS
Lamport Cells [Miller 2003]	E
SuperGlue [McDirmid and Hsieh 2006]	Java
Trellis [Eby 2008]	Python
Radul/Sussman Propagators [Radul and Sussman 2009]	MIT/GNU Scheme
Coherence [Edwards 2009]	Coherence
.NET Rx [Hamilton and Dyer 2010]	C#.NET

Cells

Cells [Tilton 2008] is a reactive programming extension to the Common Lisp Object System (CLOS). It allows programmers to define classes whose instances can have slots that trigger events when their values change. These slots are known as *cells*. Such classes are defined using the `defmodel` abstraction, which is similar to `defclass` for class definition in CLOS, but with support for defining cells as slots.

A programmer can define dependencies between cells such that when a value of one cell changes, all the dependent cells are updated. In addition, cells can get their values by evaluating rules that are specified at instance creation time. Rules contain regular CLOS code as well as reads of other cells. Rules are run immediately after instantiation and any reads to other cell slots create dependencies between the cell on which the rule is specified and the cell being read.

Computations external to the object model need to be defined as *observer functions*. One can define an observer function that is invoked when a cell of a specified name is updated to a new value. Any observer function is guaranteed to be invoked at least once during the instance creation process.

The propagation of changes in Cells is push based. That is, when a cell slot is assigned a new value, all dependent cells' rules are rerun and observers are notified to reflect the changes. The Cells engine ensures that values that do not change are not propagated, thereby avoiding wasteful recomputations. It also avoids glitches by ensuring that all dependent cells and observers see only up-to-date values. The temperature conversion example can be expressed as follows in Cells.

```
(defmodel TempConverter ()
  ((tempC :cell t
          :initform (c-in Temperature)
          :accessor tempC)
   (tempF :cell t
          :initform (c? (+ (* (^tempC) 1.8) 32))
          :accessor tempF)))
```

The `TempConverter` class has two slots (cells) `tempC` and `tempF`. The slot option `:cell t` implies that the slot is managed by the Cells engine giving it a reactive property. Regular CLOS slots that should not be handled by the Cells engine need to be specified with `:cell nil`. Slots are initialized with the `(c-in` *expression*`)` and `(c?` *expression*`)` forms. The `c-in` specifies that the cell can be modified (e.g., with the `setf` form) while `c?` specifies that a cell cannot be modified but only changes when the cell it depends on changes. In the earlier example we assume that there is a predefined `Temperature` field whose value is the current temperature in degrees Celsius that is used to initialize the `tempC` cell. A cell can refer to other cells by calling the `(^cell-name)` function, where `cell-name` is the name of another slot in the class. Therefore, `(^tempC)` implies that the

tempF cell depends on the tempC cell. Whenever the value of tempC changes, the value of tempF is recomputed.

SuperGlue

SuperGlue [McDirmid and Hsieh 2006] is a reactive language that integrates the notion of time-varying values with component programming. The basic abstractions in SuperGlue are signals, components, and rules. In SuperGlue, a reactive program is expressed as a set of signal connections between components. Components interact with each other's state through signals. A component provides state for viewing to other components through its exported interface, and views other components' state through its imported interface.

In addition, SuperGlue allows support for expressing an unbounded number of signal connections between components (i.e., the number of connections need not be known in advance). This is achieved by enhancing signals with object-oriented abstractions. It supports rules that can be used to express new connections through type-based pattern matching on existing connections. Each signal connection involves objects that reify the import being connected and the expression to which this import is being connected. The types of these objects are then used to identify the connection when rules are evaluated. The types that are used in connection pattern matching are supported through object-oriented mechanisms: nesting, traits, and extensions.

Connections between components can be identified at runtime by the types of signals to which they connect. A rule can then create a new connection relation to any existing connection that matches a specified type pattern. SuperGlue supports such type-based pattern matching with object-oriented abstractions. Thus, objects in SuperGlue serve two roles: they are containers of imported and exported signals and they serve as nodes in the program's dependency graph.

SuperGlue components are implemented using either Java or SuperGlue code but the component connections must be expressed in SuperGlue. When implemented using Java, signals are represented by special Java interfaces that enable wrapping of existing Java libraries. The evaluation model in SuperGlue is push based and the implementation ensures that glitches do not occur.

The temperature conversion example can be realized in SuperGlue as follows.

```
atom Thermometer {
    export temp : Float;
}

atom Label {
     import tempCText : String;
     import tempFText : String;
}

let model = new Thermometer;
let view = new Label;
let tempF = (model.temp * 1.8) + 32;
view.tempCtext = "Celsius: " + model.temp;
view.tempFtext = "Fahrenheit: "+tempF;
```

Thermometer is an atom that declares an exported temp signal whose value at any given point in time is the current temperature in degrees Celsius. The Label atom declares two imported signals tempCText and tempFText for displaying the values of temperature in degrees Celsius and Fahrenheit, respectively. The Thermometer and Label atoms are instantiated to create components that are then bound to model and

view. Interactions between components are established by connecting their signals together. In this example the tempF signal refers to the temp signal that is exported from the model component. Therefore, whenever there is a new value of temp, the value of tempF is recomputed. Similarly, the tempCText and tempFText signals of the view component are automatically updated to reflect the current temperature value.

Trellis

Trellis [Eby 2008] is a reactive programming library for Python that automatically manages callback dependencies. It enables programmers to express a reactive program in terms of *rules*. Rules are expressions that operate on values stored in special attributes known as *cells*. A cell value may be a variable, a constant, or computed value from a rule. Whenever a value stored in a cell changes, dependent rules are rerun. The language avoids wasteful recomputations for values that do not change.

Trellis automatically manages the order of dependencies to ensure consistent updates. During each rule execution, it keeps track of dependencies between rules and automatically rolls back in case of an error or an inconsistent result. This avoids some glitches, however programmers need to take extra care in order to write completely glitch-free programs. According to the authors, this can be achieved by dividing a Trellis program into input code, processing rules, and output rules. Input code sets Trellis cells or calls modifier methods but does not run inside Trellis rules. Processing rules compute values, while output rules send data to other systems (e.g., the display).

The temperature conversion example can be realized in Trellis as follows.

```
class TempConverter(trellis.Component):
    tempC = trellis.attr(Temperature)
    tempF = trellis.maintain(
        lambda self: self.tempC * 1.8 + 32,
        initially = 32
    )

    @trellis.perform
    def viewGUI(self):
         display "Celsius: ", self.tempC
         display "Fahrenheit: ", self.tempF
```

The preceding code snippet defines the TempConverter class that is derived from trellis.Component. A trellis.Component is an object whose attributes are reactive. The attr form creates an attribute that is writable. In this example we assume that there is a predefined variable Temperature whose value is used to initialize the tempC cell attribute. The value of Temperature at any given point in time is the current temperature in degrees Celsius. The tempF is derived from a maintenance rule that uses the value tempC to perform temperature conversion. The value of tempF is automatically recalculated whenever the value of tempC changes. @perform defines a rule that is used to perform nonundoable actions such as output I/O. In this example, a perform rule is used to display the values of tempC and tempF.

Lamport Cells

Lamport Cells [Miller 2003] is a reactive library for E [Miller et al. 2005]. In Lamport Cells, a reactive program is expressed in terms of *reactors* and *reporters*. Reactors subscribe to receive reports from the reporters, and reporters accept subscriptions and send reports to the registered reactors.

Reactors may subscribe either to receive at most one report (*whenever-reactors*) or to receive every report (*forever-reactors*). The subscription for a *whenever-reactor* lasts

only until the first report is received. The client may then decide to resubscribe when it needs a less stale value. In this respect, the evaluation model is pull based as the client re subscribes to receive a new value only when it is needed. On the other hand, a *forever-reactor* immediately resubscribes to continually receive new values. In this respect, the evaluation model is push based. The propagation of changes is unidirectional.

Lamport Cells provides support for distributed reactive programming in that the reactor and the reporter may be located on different hosts in a network. In order to provide support for distribution, subscriptions and reports are sent asynchronously with no return values. Lamport Cells tackles the potential space-leak problems by only retaining "live" references to each subscribing reactor. However, Lamport Cells does not avoid glitches (neither in a local nor a distributed setting).

The temperature conversion example can be expressed in Lamport Cells as follows.

```
def tempConverter(){
  def tempC := makeLamportSlot(Temperature);
  def tempF := whenever([tempC], fn{tempC * 1.8 + 32}, fn{true});
  return tempF;
}
```

The previous code defines the `tempConverter` function. The `makeLamportSlot` construct creates a reporter that is then bound to the variable `tempC`. In this example, the reporter is initialized with the value of the predefined variable `Temperature` whose value at any moment in time is the current temperature in degrees Celsius. The `whenever` construct creates a *whenever-reactor* that is bound to the variable `tempF`. It subscribes to the reporter `tempC` and specifies the anonymous function (defined with the keyword `fn`) that implements the temperature conversion functionality. The `fntrue` argument indicates that we automatically resubscribe. Therefore, whenever `tempC` gets a new value, its value is sent asynchronously to the reactor which in turn triggers the execution of the anonymous function in order to give `tempF` a new temperature value in degrees Fahrenheit. Both `tempC` and `tempF` can be distributed.

Radul/Sussman Propagators

Developed by Radul and Sussman [2009], Radul/Sussman propagators is a general-purpose propagation model that can be used for functional reactive programming, constraint programming, and logic programming. It is built on top of MIT/GNU Scheme. Programs are expressed as a network of *propagators* interconnected by *cells*. Propagators subscribe to cells of interest and are automatically scheduled for evaluation whenever the contents of these cells change.

Propagators and cells are the basic reactive primitives in this model. Propagators represent computations while cells are places for storing values like variables in conventional programming. Propagators continuously produce new values based on the contents of the cells while cells consume the values. But unlike variables in conventional languages where a variable holds one value, cells are specially designed to hold multiple values from different event sources at the same time. This property makes it possible to express more complex dependencies between computations.

The propagation of changes in this model is push based. The availability of new data at a cell results in triggering the execution of the propagators that previously subscribed to it. One distinguishing property of the propagators model is its support for propagating partial information. This is unlike other reactive languages where only fully computed values are propagated. Propagating partial information allows computations to perform useful computations with the currently available data. For example, in a reactive program where a certain cell is responsible for accumulating

location information (longitude and latitude coordinates), it can already propagate the longitude coordinates even if the latitude coordinates are not yet available.

Scheme primitive functions need to be manually lifted to propagators, and native values such as numbers must be manually added to cells in order to be operated on by propagators. The model itself does not ensure a glitch-free reactive program, though glitches can be avoided by use of dependency-directed tracking techniques [Radul 2009; Stallman and Sussman 1977]. Multidirectional flow of information is inherently supported by the propagation model and therefore the propagation of changes is multidirectional. The temperature conversion example in Radul/Sussman propagators is expressed as follows.

```
(define (temp-converter C F)
  (let ((nine/five  (make-cell))
        (C*9/5  (make-cell))
        (thirty-two (make-cell)))
    ((constant 1.8) nine/five)
    ((constant 32) thirty-two)
    (multiplier C nine/five C*9/5)
    (adder C*9/5 thirty-two F)))

(define tempC (make-cell Temperature))
(define tempF (make-cell))
(temp-converter tempC tempF)
```

The `temp-converter` function takes two cells `C` and `F` as arguments. It then creates three additional cells for internal computations. A cell is created using the `make-cell` function. `adder` and `multiplier` are propagators that represent the lifted versions of the Scheme primitives `+` and `*`, respectively. Propagators perform computations on the input cells and output the result to the output cell (the last argument). `constant` is a special propagator that takes a Scheme value and puts it in a cell. Attaching propagators to cells creates a propagation network that is run when the value of input cells changes. To illustrate how this works, we create the `tempC` cell that is initialized with the value of the current temperature `Temperature` in degrees Celsius. The cell `tempF` represents the output cell to which the temperature in degrees Fahrenheit is output. The expression `(temp-converter tempC tempF)` passes the two cells to the network and the temperature in degrees Fahrenheit is written to the `tempF` cell. The propagation of changes is multidirectional. Therefore, adding a new temperature value to either the `tempC` or `tempF` cell triggers the computation of the other.

Coherence

Coherence [Edwards 2009] is a language that is essentially designed for automatically coordinating side-effects in imperative programming. Coherence uses *coherent reactions* to build interactive applications. In Coherence, a coherent reaction is one in which a reaction is executed before any others that are affected by its effects.

Programmers can express dependencies between variables by way of derivations. Derivations happen lazily upon need (demand driven). Derived expressions are reevaluated every time they are accessed. To overcome the problem of wasteful recomputations, the language has some support for caching values that are still valid. Derivations are multidirectional. Changes to a variable can propagate back to the variables from which it was derived. The default semantics of Coherence is reactivity; therefore lifting of operations is not necessary.

Coherence ensures that the execution of each reaction happens before other computations that depend on it. It avoids glitches by automatically detecting the correct

execution order of reactions. Reactions are arranged in a topologically sorted order, the same technique used by FrTime [Cooper and Krishnamurthi 2006], Flapjax [Cooper and Krishnamurthi 2006], and Scala.React [Maier et al. 2010]. Similar to Trellis [Eby 2008], the entire cascade of reactions is processed in a transaction that commits them atomically or not at all. In addition, the language dynamically detects any incoherences as they occur and the effects of prematurely executed reactions are rolled back and reexecuted later [Edwards 2009].

```
temperatureConverter: {
  tempC = Temperature,
  tempF = Sum(Product(tempC, 1.8), 32)}
```

The previous code snippet defines a structure `temperatureConverter` that contains two fields `tempC` and `tempF`. In this example, we assume that there is some global variable `Temperature` that contains a time-varying value of the current temperature. The derivation of one field from another is indicated by the `=` sign. The field `tempC` is said to be derived from `Temperature` and `tempF` is derived from the expression on the right-hand side of the equals sign. The derivation expression uses the `Sum` and `Product` functions to perform the temperature conversion. Whenever `tempF` is accessed, the derivation expression is calculated.

.NET Rx

.NET Rx [Hamilton and Dyer 2010] is a reactive programming extension to .NET. It is built on top of LINQ [Microsoft 2007], a project that adds query facilities to the .NET framework. As in Scala.React, .NET Rx provides a generic interface `IObservable<T>` for representing event sources. The `IObservable` interface provides a method `Subscribe` that enables one to register a closure that will be executed when an event occurs. In addition, .NET Rx allows event composition. New specific events can be created from general events using the rich set of LINQ combinators (e.g., aggregate, selection, joins, etc.).

The propagation of events in .NET Rx is based on the push model. Consumers register interest in particular event types and then the events are pushed to them when they occur. This work is still ongoing and from the available documentation it is not explained if the language achieves glitch freedom.

The temperature conversion example can be realized in .NET Rx as follows.

```
var temperature = new Temperature();
temperature.Subscribe( temp =>
                      {
                        var tempC = temp.currentTemperature;
                        var tempF = (tempC*1.8)+32;
                      })
```

`Temperature` is a class that implements the `IObservable` interface and emits events when the current temperature changes. The `Subscribe` method registers a closure that is executed for each temperature change event. The closure includes the logic of converting the temperature from degrees Celsius to Fahrenheit.

4.3. Synchronous, Dataflow, and Synchronous Dataflow Languages

There have been programming paradigms that have been used to model (real-time) reactive systems. These include synchronous programming, dataflow programming, and synchronous dataflow programming. In this section, we give a brief review of these paradigms because there exist surveys [Benveniste et al. 2003; Whiting and Pascoe 1994; Johnston et al. 2004] that give a full review of the research on the languages

in the family of synchronous programming, dataflow programming, and synchronous dataflow programming.

Synchronous programming is the earliest paradigm proposed for the development of reactive systems with real-time constraints. Synchronous languages are based on the *synchrony hypothesis* where reactions are assumed to take no time and are atomic. It is assumed that a reaction takes no time with respect to the external environment and that the environment remains unchanged during the execution of the reaction. This assumption simplifies programs and can be compiled into efficient finite state automata, which can be translated into a program in a sequential language [Berry and Gonthier 1992]. A number of synchronous languages exist. These include Esterel [Berry and Gonthier 1992], StateCharts [Harel and Politi 1998], and FairThreads [Boussinot 2006].

Another approach that has been used to model reactive systems is dataflow programming (originally developed to simplify parallel programming) [Johnston et al. 2004; Whiting and Pascoe 1994]. A dataflow program is expressed as a directed graph with nodes representing operations and arcs representing data dependencies between computations. The difference between traditional dataflow languages and reactive languages is that dataflow languages are first order. Examples of dataflow languages include LabVIEW [Kalkman 1995] and Simulink [The MathWorks 1994].

A later development of dataflow is synchronous dataflow [Lee and Messerschmitt 1987], which combines the synchronous and dataflow paradigms. In synchronous dataflow the structure of the graph is known at compile time and can therefore be statically scheduled and converted into a sequential program that does not require dynamic scheduling. Like synchronous languages, the target domain of synchronous dataflow is reactive systems where time is a crucial element of a computation. Example synchronous dataflow languages include Lustre [Halbwachs et al. 1991] and Signal [Amagbégnon et al. 1995]. Recently, there have been FRP variations that ensure real-time guarantees (i.e., the time and space cost of each execution step for a given program are statically bound) and therefore are closer to synchronous dataflow programming languages than the FRP siblings. These include Real-Time FRP (RT-FRP) [Wan et al. 2001] and Event-driven FRP (E-FRP) [Wan et al. 2002].

5. OPEN ISSUES AND POSSIBLE SOLUTIONS

The evaluation of the reactive languages surveyed based on the taxonomy reveals some challenges that still need to be tackled in the reactive programming research. In particular, based on the taxonomy described in Section 3, very few reactive languages support multidirectionality and distributed reactive programming. In this respect, extending reactive programming to support multidirectionality and work in a distributed setting requires further investigation.

5.1. Multidirectionality

In traditional functional reactive programming, the only possible direction of change propagation is to dataflow-dependent expressions via acyclic dataflows. Some of the approaches surveyed in this article show that multidirectional constraints can be satisfied [Apt et al. 1998; Foster et al. 2007], namely Radul/Sussman propagators and Coherence. They both belong to the "cousins of reactive programming" family of systems and do not provide the classic abstractions of the sibling languages, namely behaviors and event sources.

In the field of graphics and user interaction, TBAG [Elliott et al. 1994] is a language that is more akin to functional reactive programming. It can be considered as the more graphics-oriented precursor to Fran implemented on top of C++. TBAG supports multidirectionality and has the notion of behaviors and event sources. However, TBAG

does not provide first-class behaviors and event sources. Instead, the programmer must explicitly assert the relations between the geometrical objects of a 3D scene. When operations on such objects are performed, a constraint solver attempts to satisfy all assertions by performing transformations on all related objects. One can, for example, relate two graphical objects by asserting that the distance between the two must remain fixed. Moving one of these objects results in the other one following to keep the distance constraint satisfied. If satisfying all assertions fails due to conflicts between the assertions, a runtime error is raised. For this, these objects are not mutable using standard C++ operations, but instead must be manipulated in the functional sense by performing dedicated transformations (which, like in Fran and Yampa, may be true continuous functions parameterized with time) on them that return new versions of the objects. Internally, old versions are automatically garbage collected and the latest versions are represented as time-varying behaviors. This happens by preprocessing existing graphical and mathematical operations and generating the overloaded version in the preprocessing step. This approach turned out to be satisfactory for TBAG's particular problem domain and explains the choice for Haskell as a functional language for subsequent work on functional reactive programming in Fran. With Fran, reactivity was used for a wider problem domain (i.e., more general event handling), requiring first-class behaviors and event sources and a dataflow evaluation strategy (working in a single direction).

Functional Hybrid Modeling [Nilsson et al. 2003] (FHM) is a modeling language that proposes a similar idea: instead of directly integrating multidirectionality with behaviors and event sources, it integrates multidirectionality with Yampa's signal functions and switching constructs.

Both TBAG and FHM make the constraints or relations between time-varying values explicit. This is a different approach than the one of the sibling reactive programming language where behaviors and event sources are treated as first-class values and the dependencies between them are tracked implicitly. It remains an open question whether reactive programming in the classical sense as embodied by these "sibling" languages can be reconciled with multidirectionality.

5.2. Distributed Reactive Programming

To cope with network failures and delays or to increase scalability, many distributed systems are implemented as event-driven systems. Examples are Ajax applications that communicate through asynchronous Web service requests, publish/subscribe systems, etc. An asynchronous communication style decouples the communicating parties in time: they do not have to be connected at the same time to allow communication, increasing robustness. The downside is that asynchronous invocations do not immediately return a result to the caller, but instead signal an event as soon as they have produced a response. Hence, reactive programming is very promising for more complex distributed programs that usually consist of concurrently running components that signal events to each other and react to these events by means of explicit callbacks.

There are, however, two main issues in distributing reactive programs. The first is that glitches are difficult to avoid in a distributed setting. The second is that maintaining the dependency graph in a distributed reactive program tightly couples the dependent distributed application components, and hence renders them less resilient to network failures and reduces overall scalability. We discuss both problems in the remainder of this section and look at some possible solutions and trade-offs.

5.2.1. Avoiding Glitches in a Distributed Setting. Glitches occur because of dependent code that is executed in the wrong order. This can easily happen in a distributed setting where events are communicated over the network and are hence delivered with a delay

where in the severity depends on different factors, such as the underlying network technology, network congestion, network failures, etc. Hence, timestamping of events is necessary to allow them to be correctly ordered at the receiver side. The problem here is that distributed clocks can diverge, which can be problematic for ordering events that happen in close succession.

A possible solution could be to use a centralized entity with a central clock to which all parties involved in the distributed reactive program connect and that is solely responsible for ordering events. This centralized approach of course introduces a single point of failure and possibly a considerable communication overhead as all parties involved in the system have to communicate with a single host for every event they signal and to receive every event propagated to them as well, potentially limiting scalability.

A decentralized solution could be to accept that events in close succession cannot be ordered as a fact of life and take into account a minimum time interval in which events are considered to occur simultaneously. This is similar to ideas found in real-time synchronous languages where the system is assumed to react atomically to events before any other events occur and a global clock with a minimal tick rate determines the time interval. This minimal tick rate could be used as the maximal magnitude of which distributed clocks may diverge in a distributed reactive program. As long as it can be guaranteed that all the clocks in the system do not diverge more than this time interval, glitches can be prevented while keeping a decentralized architecture. The applicability of this assumption depends on a number of factors such as the number of parties involved in the distributed interaction, the magnitude of clock divergence, the quality of the network, and, most importantly, the nature of the reactive program. For programs that have to quickly react on events occurring in very close succession this approach might not be feasible, while for programs that work on human time-scales (such as seconds, minutes, etc.) this assumption might be acceptable.

5.2.2. Network Failure Handling. Maintaining a dependency among reactive application components tightly couples these application components. Since in most cases this coupling does not have to be explicitly managed by the programmer, this is a non-issue for local applications. However, for distributed applications, a loose coupling of the communicating parties is required to achieve scalability in very large systems [Carzaniga et al. 2000] or to be applicable in certain settings such as mobile ad hoc networks [Huang and Garcia-Molina 2004]. In these cases, network failures prevent events from being propagated among distributed application components that depend on each other, causing the application to halt or causing glitches.

Publish/subscribe-style interaction offers event-driven applications an interaction style that is decoupled both in space and time [Eugster et al. 2003]. It is decoupled in time because the publish/subscribe infrastructure stores event messages such that producers and consumers do not have to be connected at the same time to propagate an event. Additionally, they are decoupled in space because dependencies are created indirectly by simply subscribing to a certain topic (topic-based publish/subscribe) or by intentionally describing the content of the events in which a subscriber is interested (content-based publish/subscribe). Hence, event producers can be dynamically replaced or be dynamically discovered when they join the publish/subscribe system and events can be multicast to a multitude of subscribers.

In an extension targeting mobile ad hoc network applications built on top of AmbientTalk/R discussed in Carreton et al. [2010], it is shown that publish/subscribe-style interaction can be integrated with distributed reactive programming, reaping the benefits of both. In the same extension to AmbientTalk/R, event messages are buffered such that they can be resent when they do not arrive to their destinations because

of network failures. Additionally, it is possible to express more complex dataflow dependencies by making use of the publish/subscribe infrastructure to notify multiple distributed parties of the same event or to subscribe to events from different producers that are aggregated in reactive lists (i.e., behaviors that denote time-varying lists of behaviors). The downside of such a decentralized approach is that glitches caused by the network cannot be prevented.

6. CONCLUSIONS

Reactive programming is a paradigm that is well-suited for developing event-driven applications which are otherwise difficult to program using conventional programming techniques. We provide a classification of reactive programming approaches along six axes: the basic abstractions for representing time-varying values, evaluation model, lifting operations, multidirectionality, support for distribution, and glitch avoidance. We discussed how several reactive languages compare to each other with respect to this classification. Most of the research on reactive programming has been carried out in the functional reactive programming area, and the representative languages have been discussed.

We have observed that the multidirectionality property is not widely supported and even not at all by classical reactive programming languages. Hence, multidirectionality seems like an interesting avenue to explore. We also observed that there is a growing interest in reactive programming to develop interactive distributed applications (such as Web applications and peer-to-peer mobile applications). This can be attributed to the fact that these applications are usually implemented in an event-driven style. Unfortunately, as we observed in Section 5, the combination of reactive programming and distribution can give rise to glitches. Hence, integrating reactive programming into distributed programming is another path that requires further investigation.

ACKNOWLEDGMENTS

We are grateful to the editors and the anonymous reviewers who made important suggestions to improve the article. We would like to thank the members the Software Languages lab for the discussion about the content of the article.

REFERENCES

Amagbegnon, P., Besnard, L., and Le Guernic, P. 1995. Implementation of the data-flow synchronous language signal. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*. ACM Press, New York, 163–173.

Apt, K. R., Brunekreef, J., Partington, V., and Schaerf, A. 1998. Alma-o: An imperative language that supports declarative programming. *ACM Trans. Program. Lang. Syst. 20*, 1014–1066.

Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Guernic, P. L., Robert, and Simone, D. 2003. The synchronous languages 12 years later. *Proc. IEEE 91*, 1, 64–83.

Berry, G. and Gonthier, G. 1992. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program. 19*, 2, 87–152.

Boussinot, F. 2006. FairThreads: Mixing cooperative and preemptive threads in c: Research articles. *Concurr. Comput. Pract. Exper. 18*, 5, 445–469.

Carreton, A. L., Mostinckx, S., Van Cutsem, T., and De Meuter, W. 2010. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS'10)*. Springer, 41–60.

Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. 2000. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00)*. ACM Press, New York, 219–227.

Cooper, G. H. 2008. Integrating dataflow evaluation into a practical higher-order call-by-value language. Ph.D. thesis, Brown University, Providence, RI, USA. https://repository.library.brown.edu/studio/item/bdr:268/.

Cooper, G. H. and Krishnamurthi, S. 2006. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European Conference on Programming Languages and Systems (ESOP'06)*. Springer, 294–308.

Courtney, A. 2001. Frappe: Functional reactive programming in java. In *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages (PADL'01)*. Springer, 29–44.

Cutsem, T. V., Mostinckx, S., Boix, E. G., Dedecker, J., and Meuter, W. D. 2007. AmbientTalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Proceedings of the 26th International Conference of the Chilean Society of Computer Science (SCCC'07)*. IEEE Computer Society, Los Alamitos, CA, 3–12.

Eby, P. J. 2008. Trellis. http://pypi.python.org/pypi/Trellis.

Edwards, J. 2009. Coherent reaction. In *Proceeding of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. ACM Press, New York, 925–932.

Elliott, C. and Hudak, P. 1997. Functional reactive animation. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*. ACM Press, New York, 263–273.

Elliott, C., Schechter, G., Yeung, R., and Abi-Ezzi, S. S. 1994. Tbag: A high level framework for interactive, animated 3d graphics applications. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'94)*. ACM Press, New York, 421–434.

Elliott, C. M. 2009. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell'09)*. ACM Press, New York, 25–36.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. 2003. The many faces of publish/subscribe. *ACM Comput. Surv. 35*, 2, 114–131.

Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. 1998. The drscheme project: An overview. *SIGPLAN Not. 33*, 6, 17–23.

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst. 29*, 3.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. 1991. The synchronous dataflow programming language lustre. *Proc. IEEE 79*, 9, 1305–1320.

Hamilton, K. and Dyer, W. 2010. Reactive extension for .net. http://msdn.microsoft.com/en-us/devlabs/ee794896.aspx.

Harel, D. and Politi, M. 1998. *Modeling Reactive Systems with Statecharts: The Statemate Approach* 1st Ed. McGraw-Hill, New York.

Huang, Y. and Garcia-Molina, H. 2004. Publish/subscribe in a mobile environment. *Wirel. Netw. 10*, 6, 643–652.

Hudak, P., Courtney, A., Nilsson, H., and Peterson, J. 2003. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming*. Lecture Notes in Computer Science, vol. 2638, Springer, 159–187.

Hughes, J. 2000. Generalising monads to arrows. *Sci. Comput. Program. 37*, 1–3, 67–111.

Jarvi, J., Marcus, M., Parent, S., Freeman, J., and Smith, J. N. 2008. Property models: From incidental algorithms to reusable components. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, New York, 89–98.

Johnston, W. M., Hanna, J. R. P., and Millar, R. J. 2004. Advances in dataflow programming languages. *ACM Comput. Surv. 36*, 1, 1–34.

Kalkman, C. 1995. Labview: A software system for data acquisition, data analysis, and instrument control. *J. Clinical Monitor. Comput. 11*, 1, 51–58.

Lee, E. A. and Messerschmitt, D. G. 1987. Synchronous data flow. *Proc. IEEE. 75*, 9, 1235–1245.

Maier, I., Rompf, T., and Odersky, M. 2010. Deprecating the observer pattern. Tech. rep. http://lampwww.epfl.ch/~imaier/pub/DeprecatingObserversTR2010.pdf.

Mcdirmid, S. and Hsieh, W. C. 2006. SuperGlue: Component programming with object-oriented signals. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*. 206–229.

Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A., and Krishnamurthi, S. 2009. Flapjax: A programming language for ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*. ACM Press, New York, 1–20.

Microsoft C. 2007. LINQ: NET language-integrated query. http://msdn.microsoft.com/library/bb308959.aspx.

Miller, M., E. Tribble, D., and Shapiro, J. 2005. Concurrency among strangers: Programming in e as plan coordination. In *Proceedings of the Symposium on Trustworthy Global Computing*. Lecture Notes in Computer Science, vol. 3705, Springer, 195–229.

MILLER, M. S. 2003. The reporter/reactor pattern. http://www.erights.org/javadoc/org/erights/e/elib/slot/EverReporter.html.

NILSSON, H., PETERSON, J., AND HUDAK, P. 2003. Functional hybrid modeling. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL'03)*. Springer, 376–390.

ODERSKY, M., ALTHERR, P., CREMET, V., EMIR, B., MANETH, S., ET AL. 2004. An overview of the scala programming language. Tech. rep. IC/2004/64, EPFL Lausanne, Switzerland.

ORACLE. 1997. JavaBeans component model. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html.

PETERSON, J. AND HAGER, G. 1999. Monadic robotics. In *Proceedings of the 2nd Conference on Conference on Domain-Specific Languages (DSL'99)*. Vol. 2, USENIX Association, Berkeley, 8–8.

PETERSON, J., HUDAK, P., REID, A., AND HAGER, G. D. 2001. Fvision: A declarative language for visual tracking. In *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages (PADL'01)*. Springer, 304–321.

PUCELLA, R. R. 1998. Reactive programming in standard ml. In *Proceedings of the International Conference on Computer Languages (ICCL'98)*. IEEE Computer Society, Los Alamitos, CA, 48–57.

RADUL, A. 2009. Propagation networks: A flexible and expressive substrate for computation. Ph.D. thesis, MIT. http://web.mit.edu/~axch/www/phd-thesis.pdf.

RADUL, A. AND SUSSMAN, G. J. 2009. The (abridged) art of the propagator. In *Proceedings of the International Lisp Conference (ILC'09)*.

SCULTHORPE, N. 2011. Towards safe and efficient functional reactive programming. Ph.D. thesis, Nottingham, UK. http://www.ittc.ku.edu/~neil/papers_and_talks/thesis.pdf.

SPERBER, M. 2001a. Computer-assisted lighting design and control. Ph.D. thesis, University of Tbingen. http://tobias-lib.uni-tuebingen.de/dbt/volltexte/2001/266/.

SPERBER, M. 2001b. Developing a stage lighting system from scratch. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York, 122–133.

STALLMAN, R. M. AND SUSSMAN, G. J. 1977. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artif. Intell. 9*, 2, 135–196.

STEELE, JR., G. L. 1980. The definition and implementation of a computer programming language based on constraints. Tech. rep., Cambridge, MA, USA. http://dspace.mit.edu/handle/1721.1/6933.

THE MATHWORKS. 1994. Simulink - Simulation and model-based design. http://www.mathworks.nl/products/simulink/index.html.

TILTON, K. 2008. The cells manifesto. http://smuglispweeny.blogspot.com/2008/02/cells-manifesto.html.

WAN, Z. AND HUDAK, P. 2000. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. ACM Press, New York, 242–252.

WAN, Z., TAHA, W., AND HUDAK, P. 2001. Real-time frp. *SIGPLAN Not. 36*, 10, 146–156.

WAN, Z., TAHA, W., AND HUDAK, P. 2002. Event-driven frp. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL'02)*. Springer, 155–172.

WHITING, P. G. AND PASCOE, R. S. V. 1994. A history of data-flow languages. *IEEE Ann. Hist. Comput. 16*, 4, 38–59.

ZABIH, R., MCALLESTER, D., AND CHAPMAN, D. 1987. Non-deterministic lisp with dependency-directed backtracking. In *Proceedings of the 6th National Conference on Artificial Intelligence*. AAAI Press, 59–64.

Received May 2010; revised December 2011, June 2012; accepted August 2012