A Report on KScript and KSWorld

Yoshiki Ohshima, Bert Freudenberg, Aran Lunzer
Ted Kaehler

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Research Note RN-2012-001

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

A report on KScript and KSWorld

Yoshiki Ohshima Bert Freudenberg Aran Lunzer
Ted Kaehler

October 13, 2012

1 Introduction

We are interested in making an end-user oriented authoring environment. In such an environment, the user can compose objects to construct various kinds of interactive applications from simple toys to prototypes of user interfaces with direct manipulation, and even *real* applications.

One could imagine that what we want is something like Hypercard; but we would like to support a very uniform object model and infinite embedding of objects into each other. There should be no system-defined widgets; a button object would be just like any other graphical object except having behavior that the user can change and customize. Also, by allowing objects to be embedded into each other, the user can make his own composite objects so that he can work with better abstractions.

One also could imagine that it is something like Etoys. However, we would like to support a better model of time in the programming language. It would allow that managing the rate of execution (such as slowing down the system or even go backward in time) can be done easily. Also we would like to support mixing and matching of objects. Time-aware computation models have a long history that goes back to John McCarthy's Situation Calculus [14] and still work is going on such as Dedalus [2], but they have not been fully adapted into Etoys-like authoring environments.

We also would like to contribute to the goals of the STEPS project [12]; namely, we would like programs written for this framework to be compactly written and intention revealing.

Based on these observations, the key idea we decided to look at was "reactive programming". Reactive programming works similar to spreadsheets: The definition of a variable, or *formula*, refers to other variables, and when any of the input variables (the *dependency*) changes, the variable that depends on them (the *dependent*) is updated. Because the dependency-dependent relationship is transitive, the update ripples through the network of dependencies. In our current implementation, we follow the formulation of Functional Reactive Programming (FRP) [8] for the distinction between behaviors and events, and draw upon Flapjax [15] for the naming of combinators.

VPRI Research Note RN-2012-001

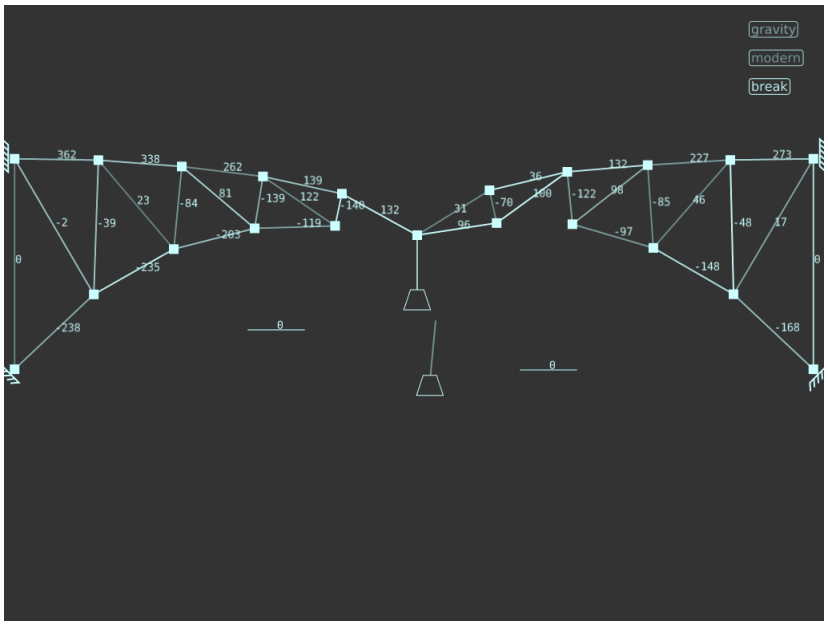

Figure 1: Simulation of a bridge done in (the predecessor of) the KSWorld.

This is a good match because a large part of programming a graphical user interface (GUI) is writing code that handles some changes in objects and dealing with time-based actions. FRP can provide a good model for this.

In a GUI framework, we would still like to have the abstraction of "objects" for representing graphical entities. Reactive programming can be naturally extended to incorporate the concept of objects by allowing a field of an object to be a reactive variable and also allowing variables to hold an object.

A notable system in the past that shares many of the above ideas was Forms/3 by Burnett et. al. [7]. Compared to Forms/3, our system provides higher-level organization such as objects and embedded graphical objects to support applications and projects that are much bigger. Also, more importantly, our system aims to be **self-sustained**. The editors, inspectors, etc. used in the authoring system would be written on top of the same system (eventually).

Another important idea we decided to incorporate is that the references (names) in a formula are treated as keys to perform late-bound resolution. This allows the system to be organized as a loosely-coupled set of objects. With it, forward referencing to an object that may not exist yet, and marshaling an object out of a group of inter-related objects becomes much easier.

As an end-user authoring system, we would like to support exploratory-style programming. This would require the ability to reformulate things in a way that cannot be planned in advance. For example, the user may write scripts that evolve over time or he may even decide to create his own meta-tools for existing objects. Late binding helps to support such ideas.

1.1 KSWorld and KScript

Based on these ideas, we created a graphical user interface framework called *KSWorld*, which is written in a new language called *KScript*.

The system and language have the following characteristics:

Dictionary-like objects The base language of KScript resembles CoffeeScript. KScript provides simple object-oriented language features such as methods and fields in objects. An object in KScript is a variable-length dictionary similar to JavaScript. This dictionary-like nature is used as the basis for variable name resolution.

FRP-style dataflow programming The base language is extended to support FRP-style dataflow programming. Such a dependency description creates an object that represents a stream of values called the "event stream", or simply the "stream". A stream typically depends on other streams, and a new value for the stream is computed reactively when the conditions in the dependency streams are met. In other words, the streams form a network and the change of a stream value propagates to its transitive dependents over the network.

Time-based streams operate in a pseudo (logical) time domain, decoupled from real time.

Late-bound variable resolution When a formula is defined for a stream, only the names used in the formula are registered as keys to look up dependencies. At runtime, the actual streams corresponding to the dependencies' names are resolved in the object that is the owner of the stream (i.e., it serves as the namespace). If a specific binding in the dependency network has changed in the interim, the formula can still work.

Setting the initial value, obtaining the previous value It is convenient to regard some streams as representing a continuous value over time (in FRP terminology they are called `behaviors`, as opposed to `events`, which are discrete). There is a language construct to create a behavior.

Also, there is a language construct to obtain the previous value of a stream. (Essentially, this is the same as `pre` in Lucid, or `earlier` in Forms/3). This allows the author to express certain problems in a more straightforward manner.

GUI Framework There is an accompanying GUI framework written from ground up. The GUI Framework handles very basic features such as low-level event routing and rendering (although not written in the dataflow language). KScript and the dataflow extension uses the framework to provide a dynamic environment.

We have written simple and complex examples in KSWorld (and its predecessors). For example, the code for a button widget is about 15 lines in

the dataflow language. There are fairly large examples written in the system; figure 1 shows one such example.

The remainder of this article is organized as follows. Section 2 describes the language of KScript, that has a base object-oriented language and the dataflow extension. Section 3 describes the main combinators in the dataflow language. Section 4 describes the KSWorld GUI framework that is built in KScript. Section 5 shows how a simple widget can be implemented, and shows the examples of complex applications. Section 6 shows the basic strategy of the implementation. Section 7 shows other possible paths we could have taken. (This may be the most valuable part.)

2 KScript Language

This section describes the language of KScript.

2.1 Base Language

The surface syntax resembles CoffeeScript [5]. We wanted reasonably clean syntax with a straightforward object model to experiment with the key concepts.

The original CoffeeScript inherits some problems from its ancestor (JavaScript), such as the distinction between a single arrow `->` and a fat arrow `=>` in defining a function. We eliminated such issues and simplified the language.

The objects in the language are dictionaries. To access fields, you can use the regular dot and square bracket notation (`.` and `[]`), as well as an at-sign notation (`@`). Such a dictionary-like object is called KSObject.

2.2 FRP-style Data flow extension

On top of the base language, an FRP-style dataflow extension is added. When an expression is enclosed in a double dollar quotation (`$$(...)`), the expression is treated as a definition of a dataflow graph. The result of evaluating the expression is a kind of object called the "event stream" or "stream", and it typically is assigned into a property of an object. For example, the following creates a stream that updates itself with a new value at every 200 milliseconds with a combinator called `timerE()`, and the stream is assigned to a property called `timer`. The `timer` stream's new values are integers in increments of 200, starting from the "logical time" (more on this later) when the timer was created.

```
this.timer = $$(timerE(200))
```

Then, the stream can be used by other streams:

```
this.fractionalPart = $$(@timer % 1000)
this.sounds = $$(FMSound._pitch_dur_loudness(@fractionalPart, 0.2, 100))
this.player = $$(@sounds.play())
```

The values in the `fractionalPart` stream are the milliseconds part of the `timer` stream as `%` is the remainder operation ; i.e., the values are the sequence of [0, 200, 400, 600, 800, 0, 200, ..., 800, 0, ...]. The `fractionalPart` obtains a new value for every 200 milliseconds, because it is reacting to the change in `timer`.

Then, the values are used by the `sounds` stream that creates `FMSound` objects with specified pitch and 0.2 second as duration (and 100 for loudness). The stream of sounds are in turn `play()`'ed by the `player` stream.

An important concept is that the "at-sign name look up" in a stream definition (such as `"timer"` in `fractionalPart` and `"fractionalPart"` in `sounds`) means that the actual stream bound to the name is looked up *each time* the dependencies are tested for the update. The look up is done in the KSObject that owns the stream.

Imagine that this is used in an end-user authoring system. The user explores and modifies scripts even while the timer is actively ticking. As in any live-editing environment, the user may want to keep modifying such scripts without stopping the timer and without going through the cycle of compiling and restarting the program. To allow such flexibility, the streams need to be *loosely-coupled*; that is, a stream should not hold onto any direct pointers to other streams but rather just remember them with symbolic variable names so that the system can swap the dependencies for a stream without any problems.

The stream definition can be more complex than a single expression. You can construct a compound stream from many sub-streams.

In the general case, when a dependency acquires a new value, the dependents compute their new values based on the new value. However, sometimes the programmer wishes to explicitly stop propagating the computation downstream. The value `undefined` is used for this purpose; when the dependency's new value is `undefined`, the system does not cause any update in its dependents. For example the value in `timerViewer` stream below does not exceed 10.

```
this.stoppingTimer = $$(if @timer > 1000 then undefined else @timer)
this.timerViewer = $$(@stoppingTimer / 100)
```

The expression within the $$(...) quotation is compiled differently. The quotation is like the `{!...!}` quotation of Flapjax, but here we use a legal function call syntax with a special function name so that the parser of KScript does not have to be aware of the data-flow extension.

The compilation scheme is described in section 6. The important point is that the resulting stream objects in general do not contain any pointers to other stream objects. The advantage of not having any pointers among streams is very important. As mentioned earlier, changing the dependency graph requires no extra bookkeeping effort. Garbage collection also works without any trouble of unregistering dependents from their dependencies.

2.3 Behaviors and Events

In FRP, there is a distinction between "behaviors", which represent continuous values over time, and "events", which represent sequence of discrete changes over time.

A behavior can be easily converted to events and vice versa (a behavior is like a stream of events but the value of the last event is cached and used as the current value of the behavior; an event is like a behavior but the change in the current value is recorded as a change event). Especially because KScript is a dynamically typed language, no the type constraint on them is enforced; one could mix them in the program.

But it still helps to think of the distinction. Most notably, behaviors have initial values thus they have always values while events don't have any value until the first change occurs. The way KScript defines behaviors is to provide the initial value and increment either by using the keyword "`fby`" (meaning 'followed by' and borrowed from Lucid), or to send `startsWith()` (borrowed from Flapjax) to a stream to give the initial value.

2.4 Setting Values into Streams

To leave room for exploration and tools such as inspectors, the notion of "setting" or "assigning" a value into a stream exists in the KSWorld. For example, as shown in the examples of section 4.1, the stream of values that represents the position of a `Box` cannot be purely defined by a function of other streams, as one may want to move the `Box` via a halo, an inspector or other external means. (In a self-sustaining system that KSWorld is aspiring to, different kinds of inspectors may be created after the `Box` was created.)

To support such actions, a stream supports an operation called `set`. The `set` operation simply takes an argument and stores it into the stream. The dependents of the stream will be evaluated in the next time step.

This concept meshes nicely with the "constant stream", which has no dependency and just represents a constant value. However, nothing would be totally constant in a self-sustaining system. Again, for the example of the stream that represents the position of a `Box`, such a `Box` is typically born as "vanilla", meaing with no behavior attached, thus its position would be considered constant. However, such a `Box`'s position sometimes has to be changed to support the notion of direct manipulation and editing from a meta tool. Such manipulation can be easily supported by setting new values into the "constant" stream, and the dependents of the position can be naturally resolved by the same dependency evaluation mechanism.

We call such a constant stream (but that still supports `set`) a "value stream". To create such a stream, there is a method called `streamOf()` for `EventStream`. It takes one argument and creates a constant stream with the argument or, in other words, creates a value stream with the argument as the initial value.

To evaluate a group of streams, there is an object called `Evaluator`. There is a method for `Evaluator` to add streams from a given object (called `addStreamsFrom()`)

```
Evalator.addStreamsFrom = (anObject) ->
   for stream in anObject
      // add stream to the list of streams called streams

Evaluator.sortAndEvaluateAt = (pseudoTime) ->
    var sorted = this.topologicallySortedteams()
    for stream in sorted
        stream.updateIfNecessary(pseudoTime)
```

Figure 2: The evaluation method of KSObject in pseudo-code

and a method to evaluate the set of streams in the proper order. That is to say, streams are sorted topologically, and then each element in the sorted list will be updated when any of its dependencies triggers it.

3 Combinators

KScript offers several "combinators" that let you combine other streams to build up the network of streams. The combinators' names and functionality are drawn from FRP implementations, especially Flapjax.

3.1 `mapE`

The most common combinator is `mapE()`, whose syntax looks like a message send. As its argument it takes a one-argument function, and it is "sent" to a source stream as its receiver. Namely, its type signature is:

```
recv.mapE(func) :: EventStream b
recv :: EventStream a
func :: a -> b
```

The values of the resulting event stream are the results of applying the function to the values from the source stream. However, it is often the case that the side-effects caused by the function evaluation are more important than the actual values.

Remember that the source stream is looked up from the object's dictionary with the `@`-syntax. A typical use of `mapE` is:

```
this.color = $$(... "a stream definition"...)
this.update = $$(@color.mapE((c) -> ... c ...))
```

where, "`color`" is a key in the owning object, and the value for the key is the stream. The second line states that whenever the current value in the stream bound to `color` changes, the function is evaluated with the value as the argument, and the result from the function becomes the new value of `update`.

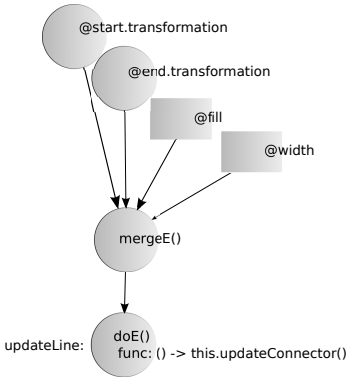

Figure 3: an exmaple of `mergeE` in section 3.3

3.2 `doE`

`doE` is similar to `mapE` but takes a 0-ary function as its argument:

```
recv.doE(func) :: EventStream b
recv :: EventStream a
func :: _ -> b
```

As in `mapE`, there are cases where the side-effects caused by the function can be computed without using the source value.

3.3 `mergeE`

`mergeE` combinator takes one or more objects as its arguments, and whenever any of them changes, the latest value becomes the `mergeE()` stream's value. Namely, its type signature is:

```
mergeE(ev_1, ..., ev_n) :: EventStream a
ev_1 ... ev_n :: EventStream a
```

Note that the type constraint `a` is not actually enforced, as this is in a dynamically-typed language.

Again, it often is the case that the actual value of `mergeE` does not matter, but only the fact that something is to be updated does. For example, imagine you have a line segment object (called a "Connector") in an interactive sketch, and it has to update its graphical appearance whenever one of its end points (`start` and `end` bound in the `this` dictionary) moves, or its line style such as width or fill changes. For this to work, we would just merge the dependencies and then invoke a method with side-effects (`updateConnector()` below) that recomputes the graphical appearance:

```
this.updateLine = $$(mergeE(
        @start.transformation,
```

```
@end.transformation,
@fill,
@width).doE(() -> this.updateConnector()))
```

Here, two streams that represent the transformation of two `Box`es that represent end points, the `fill` property, and the `width` property are merged. When any of the inputs changes, the `mergeE` stream itself changes and triggers the function passed to `doE`. (See figure 3.)

3.4 `collectE`

`collectE` combinator creates a stream that depends on the source event stream and takes an initial value and a two-argument function. When the source stream acquires a new value, the new value and the old value are passed to the function and create a new value.

Namely, its type signature is:

```
src.collectE(init, func) :: EventStream b
where
    src  :: EventStream a
    init :: b
    func :: (b, a) -> b
```

`collectE` creates a "stateful" stream as its values potentially depend on the series of previous values. A typical example of `collectE` is to accumulate all incoming values in some way, but the other use in the system is to hold onto the old value to represent the change from the last value.

`collectE` can also be considered a simple way to specify a direct circular dependency.

3.5 `update`, and access to previous values

The syntax of `update` differs from the combinators described so far. It is created with the `fby` construct and creates a behavior in the FRP sense.

`update` is similar to `collectE`, in that it takes an initial value and an expression to calculate new values:

```
this.sliderValue = $$(0 fby @sliderValue2D.y())
```

The stream called `sliderValue` starts with 0 but whenever the value of stream `sliderValue2D` changes, it acquires a new value (which is the y-component of `sliderValue2D`). The difference from `collectE` is that it can have more than one dependency as the expression can comprise references to two or more streams and values.

This ability causes an interesting case with circular dependency. A stream can refer to itself by explicitly using the "earlier value". When the prime mark (`'`) is attached to the variable name, the previous value of the referred stream is used to compute the new value. For example,

```
this.nat = $$(0 fby @nat' + 1 on @timer)
```

The stream `nat` recomputes its new value whenever the `timer` stream gets a new value, and the new value is its previous value incremented by 1. It also allows mutually dependent multiple streams. For example, one could write:

```
this.a = $$(false fby !@b' on @timer)
this.b = $$(true  fby !@a' on @timer)
```

where `!` represents a logical negation operator. Suppose both streams are being evaluated at time t. Neither stream sees the other stream's value computed at t. In effect, the previous values of each are used simultaneously to compute the new value.

The `'` mark is similar to the `pre` construct in Lucid Synchrone [18], or `earlier` of Forms/3.

If the above code is written without `'`:

```
this.a = $$(false fby !@b)
this.b = $$(true  fby !@a)
```

the result will be unspecified; the circular dependency between `a` and `b` causes a race condition and taking the "current value" could mean before or after the update.

3.6 switchColE

There is a special purpose combinator `switchColE`. In GUI programming, it is often the case that there is a collection of homogeneous objects, and a need for detecting a change of the stream under the same name in any of the objects. For example, a menu is a collection of buttons, and when the `fire` stream of any of the elements is updated (due to the user clicking), the menu reacts to it. For example:

```
this.items = col       // a collection of buttons
this.itemSelected = $$(switchColE(@items, "fire"))
```

where `this` is the handler for the menu, and variable `col` is a collection of buttons. The `switchColE` stream looks for a change in `fire` in any item in the `items` collection and uses it as its new value.

3.7 timerE

This combinator was used in the example in section 2.2. It looks like a function call and takes a numeric argument and creates a stream that updates itself for the number of millisecond interval.

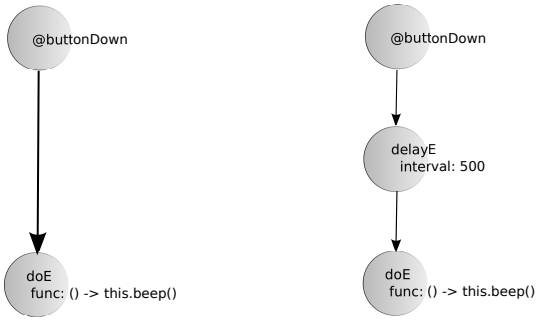

Figure 4: an example of `delayE` inserted into a pipeline

3.8 `delayE`

`delayE` delays the propagation of events for a specified length of time. The syntax of `delayE` looks like a message send. It takes a numeric argument and the events in the receiver will be delayed the specified number of milliseconds. For example, imagine a stream definition:

```
this.myBeeper = $$(@buttonDown.doE(() -> this.beep()))
```

where `@mouseDown` is a stream that gains a new value when the mouse button is pressed, and `beep()` makes a noise. In effect, `myBeeper` represents a pipeline with two nodes (`buttonDown` and `doE`) that makes a beep when the mouse button is pressed.

Now, if you insert `delayE` into the pipeline (also see figure 4):

```
this.myBeeper = $$(@buttonDown.delayE(500).doE(() -> this.beep()))
```

each event from `buttonDown` is delayed for 500 milliseconds before triggering the function passed to `doE`.

4 KSWorld: the GUI framework

KSWorld is a GUI framework designed to work with KScript. The overall structure is drawn upon Morphic [9], Lessphic [17], and Tweak [19]. The framework maintains the display-scene tree that consists of graphical objects called `Box`es. The display scene represents a 2.5 dimensional scene, and the framework renders the `Box`es onto the screen.

We take the idea of uniform object model seriously; even the individual characters in the text field and button labels themselves are just like any other `Box`es, except their graphical appearances ("shapes", described below) resemble the glyphs.

We need to have a way to specify the actions of `Box`es. The actions in essence are what the box does in reaction to certain kinds of events, such as raw user events, timer events, and all transitive causal effects. This idea, handling the

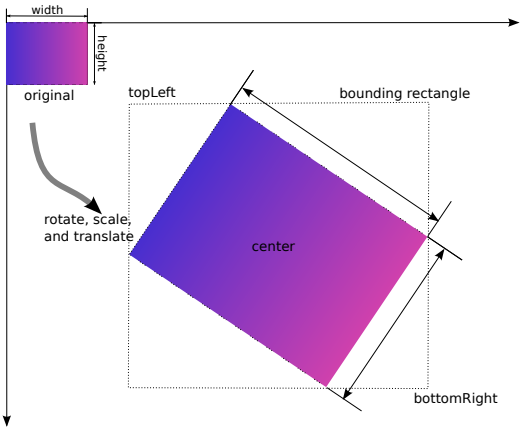

Figure 5: A `Box` in its coordinate system and in the container's coordinate system

transitive cause and effects, suggests that the FRP-style dataflow model is a good match.

Because the `Box`es are just a kind of KSObjects and they can just hold onto the streams in their fields, specifying behavior is merely assigning streams to the fields.

Note that some parts of the framework are not written in the functional reactive style, because some parts are inherently stateful and non-pure. Imagine that you have two boxes both having a mouse-over behavior attached. The functional description of the behavior for each box would be: "when the mouse pointer is within my bounds, then react to it in a particular way", indifferent to the existence of the other box. However, when these two boxes' bounds overlap, the result we'd like is not a sum of two. What is expected is that the box in front of the other reacts to the event, but not the one behind, even though the mouse pointer may be in its bounds. Essentially, routing events requires some ordering in evaluation.

4.1 Boxes

Each `Box` manages its fundamental properties and derived properties as streams. Fundamental streams include `shape`, which represents the bezier curves and their fills, `transformation`, which is its transformation matrix relative to its container, and a few flags such as whether the box clips its contents and whether the box is visible. One of the derived streams is `extent`, which consists of width and height and depends on `shape` and represents the width and height of the box in its own coordinate system. Another is `bounds`, which depends on `transformation` and `shape` and represents the enclosing bounding rectangle in its parent coordinate system. See figure 5. The original box is scaled, rotated, and translated. The width and height is still the same, as it is in its coordinate system while the bounding box is the dotted box enclosing the transformed box. The bounding rectangle defines some other values such as `topLeft`, `center`, `bottomRight`, etc.

As these streams need to have values at all times, they are defined as behaviors in the FRP sense.

Since a fundamental property such as `transformation` is just a value stream, the programmer could overwrite the `transformation` field with its own definition. For example, if you want to make the offset of the `Box` depend on a timer, you could write the following to move the box based on the timer's value (divided by 20):

```
this.transformation = $$(MatrixTransform2x3
                            .withOffset(P(timerE(100)/20, 10)))
```

This works fine, but also this makes a "too strong statement". Imagine this is done in an authoring environment like Etoys. The user would try to experiment with different speeds or directions of movement by changing these constant values ("100", "20", etc.), or even making them variables. But most notably, sometimes the user would want to stop the movement of the `Box`. If the `transformation`'s definition had been changed as above, it would be harder to turn on and off the timer without messing things up.

To maintain the possibility of such future changes, we usually use the accessor method for `transformation` called `transformation()`. There is also a wrapper to change only the offset part of the `transformation` called `translation()`. With the `translation()` method, one can write the equivalent code as follows:

```
this.myMover = $$(timerE(100)
                     .mapE((t) -> this.translation((P(t/20, 10)))))
```

This basically gives you the same effects above. The latter does look a bit more convoluted (by having the combinator `mapE()`), but allows some more flexibility. For example, the user can just nil out the `myMover` property to stop the movement. (This is not easy in the above setting, because `transformation` is an essential property used by event routing and rendering, so you just cannot nil it out.)

To describe actions among multiple objects, the objects are referred to by names from a namespace object. Any `Box` in the display scene can be a namespace. So one could write:

```
this.otherBox = @container.first()
this.myAligner = $$(@otherBox.bounds.mapE((bb) ->
                      this.topLeft(bb.topLeft())))
```

to keep the top left position of `this Box` to be equal to that of the `Box` bound to the `otherBox` property (which, in this example, is initialized with the first element of the containing box).

For examples such as buttons and color pickers see section 5.

4.2 More Accessors

It is convenient to have elaborated ways of specifying the geometric property of a `Box`. For example, to specify the location of a `Box`, one would set the corner position, center, or mid-point of an edge, etc. In the current implementation, there are a set of accessors for this purpose. These include `topLeft()`, `topRight()`, `center()` and so on.

Also, it is convenient to specify the transformation of a `Box` in the differential form; e.g., `translateBy()`, `scaleBy()`, and `rotateBy()` to specify the difference from the current transformation.

There is an interesting issue around this. Please see section 7.3 for more detail.

4.3 User Event Routing

When the framework receives a user event (such as "button down", "key up", etc.), the framework traverses the display scene in depth-first manner to determine the receiving box of the event. From the event, its "event type name" (such as `buttonDown`, `keyUp`) is assigned. The recipient is either the box designated as "focus", or the first box found during the traversal whose bounding box includes the coordinate of the event and has a stream with the event type name. When the box is found, the event is `set` to the stream as the new value.

Now, the same dependency resolution mechanism can evaluate the dependencies of such streams to cause the reactions to the events.

In `registerEvents()`, the raw-events delivered to the system are routed to an appropriate widget.

After finishing `registerEvents()`, the display tree is traversed to evaluate the dependency graph that is triggered by the raw-events and timers (`withAllBoxesDo()` applies the given function to all sub boxes). As described below, the dependency graph can be different from one display cycle to another. We just sort them at every display cycle (with a simple caching scheme). The argument for `sortAndEvaluateAt()` is a number that represents the time when the streams are to be evaluated. The `mapTime()` method maps the phyical time

```
while true
    var currentTime = getMilliseconds()
    registerEvents()
    evaluator = Evaluator.new()
    window.withAllBoxesDo((box) ->
        evaluator.addStreamsFrom(box))
    evaluator.sortAndEvaluateAt(this.mapTime(currentTime))
    window.draw()
```

Figure 6: The top-level loop of KSWorld in pseudo-code

```
while true
    var currentTime = getMilliseconds()
    registerEvents()
    for root in window.selectAllTimeBubbles()
        evaluator = Evaluator.new()
        root.withAllBoxesInThisTimeBubbleDo((box) ->
            evaluator.addStreamsFrom(box)
        evaluator.sortAndEvaluateAt(root.mapTime(currentTime))
    window.draw()
```

Figure 7: The top-level loop of KSWorld in pseudo-code with time bubbles

obtained by the `getMilliseconds()` function to a logical-time that the `this` object maintains. (More on this in the next section.)

4.4 Time Bubbles

The time the streams deal with is the "logical time" (or "pseudo time"), which is decoupled from wall-clock time. However, having only one timeline is often too limiting. For example, imagine that one would like to create an event-playback mechanism (similar to the Event Recorder in Squeak), where an embedded widget is showing a slow-motion playback of the past interaction. The chrome of the player needs to be operating in wall-clock time, so that user interactions such as stopping the playback are handled in the normal way, while the timeline in the embedded widget is on its own.

To allow different timelines in different set of graphical objects, we introduced the concept of "time bubbles". Some `Box`es in the display tree can be designated as the root of a time bubble. Such a `Box` can have its own `mapTime()` method and can provide a different value for the evaluator, thus control the time within the sub-tree (or bubble) in the display scene.

Figure 7 shows the top-level loop of KSWorld with the time bubble concept incorporated (which is closer to the real version; see section 7.6 for one more version). Instead of simply adding all streams of all boxes to one evaluator, an

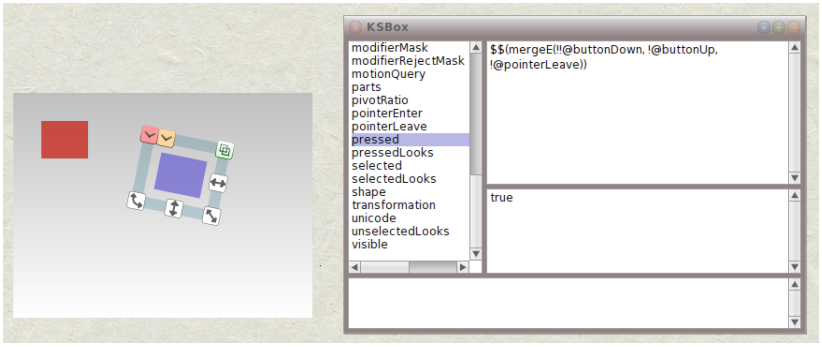

Figure 8: An external inspector looking at a `Box`

evaluator is created for each time bubble. To evaluate the time bubble, the roots of time bubbles are identified first (`withAllBoxesInThisTimeBubbleDo()`), and then for each root, the boxes under it (but above any other time bubbles) are added to the evaluator (`withAllBoxesInThisTimeBubbleDo()`), and the streams are evaluated for the time bubble.

4.5 Rendering

KSWorld uses the Gezira graphics engine for rendering. On top of the core Gezira engine [3], there is a canvas abstraction that supports transformation and clipping. The `shape` objects of all `Boxes` are traversed and rendered onto the canvas.

In previous versions, we have attempted elaborated damage-region management to avoid excessive drawing. In the latest version we are using OpenGL to composite Gezira-rendered bitmaps, and have a box-level bitmap caching mechanism. Gezira is asked to render the shape to bitmap only when the box invalidates the cache.

4.6 Layout Mechanism

One important feature of GUI frameworks is the layout. One could wish to write layout as a relationship between `Boxes`' bounds and specify it in the FRP; this again is possible and we have tried for a limited extent. However, besides the fact that such a relationship among `Boxes` would inevitably have a circular dependency, the kind of applications we are interested in have very dynamic nature; some of the `Boxes` may have time-driven or user-driven actions. One cannot set up a meaningful dependency network among `Boxes` that places them properly.

Based on this observation, we instead attach a layout object to a containing `Box` if necessary. After processing all streams, the layout objects in the display scene are asked to lay out the contents of the corresponding `Box`. The layout

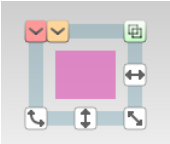

Figure 9: The Halo in KSWorld

causes changes in the `transformation` of `Box`es. Such changes will be picked up in the next evaluation cycle.

There are a few types of layout objects. There are vertical and horizontal layouts that lay contents boxes vertically or horizontally. There is a constraint-based layout. The programmer can specify the relationships among boxes declaratively, and the layout works as a solver to determine the `Box`es' positions. The text layout is one such layout. Borrowing from the ideas in our past frameworks [4], each character in a text is also a `Box`, and the text layout's job is no different from other kinds of layout; namely, place the contents `Box`es appropriately.

4.7 Halo

In KSWorld, visual objects are manipulated using a halo mechanism [11]; see figure 9. It is a highlight around a `Box` that shows that the `Box` is selected, and it allows the user to move, rotate and resize the `Box` without invoking the `Box`'s own actions (such as any button-click action it might have). At the same time, when actions of the `Box` are changing the `Box`'s transformation programatically, the halo tracks such changes and repositions itself. As it exists within the uniform object model, the halo itself consists of the same kind of `Box`es.

However, there is a circular dependency between the `transformation` of the halo and that of the target box. When the halo moves the target should be moved. But when the target moves, the halo should be moved. Even with the loose coupling between the target and halo (the halo can operate on any object stored in the "haloTarget" stream in the top-level `Box`), such actions cannot be expressed properly in a dataflow language.

For this reason, the `transformation` and the `extent` of a halo are set in the same way the layout works: when the user drags the halo, the target box's `transformation` is set via the accessor described above. When the target `Box` transformation is changed, the halo moves at the layout stage.

4.8 Window, World State and Hand

There are two `Box`es that receive special support from the framework. One of them is the "Window", which represents the top-level `Box` in the display tree. Besides being the top node in the tree, it maintains global system information that is packaged into a separate object called the "WorldState". The WorldState works as a liaison between the hosting environment and the KSWorld.

Another object is a "Hand", that is the abstract representation of the user input device. It also is just a `Box` in the display scene and its position is usually correlated with the location of the pointing device cursor. We currently do not support multiple pointing devices, but Morphic implemented such a feature just by having an array of Hands.

There is a special property name (`__topContainer__`) to access the `Window` from any `Box` in the display tree. In the following code,

```
this.myTicker = $$(@__topContainer__.globalClock.mapE((time) -> ...))
```

the `globalClock` property of the World is checked for updates, and if it has a new value, the function argument for `mapE` will be evaluated. To build an application that has many dependents from a single ticking clock in the system, one could define it at the Window in this manner.

4.9 External Inspector

As of writing this article, there is an external tool (i.e., not running in KSWorld) to inspect the streams and their values and make changes to the definitions and values. (See Figure 8.) In the near future, we would like to implement such a tool in KSWorld itself.

5 Examples

In this section small examples like a simple button, to a complex example such as an application of bridge simulation are described.

5.1 The Button Explained

The goal of the KSWorld is to describe not only applications on top of a host GUI framework with pre-made widgets, but also to be able to write such widgets and customize them. We begin with the button widget as an example. It is also a good example to see how much code we need for a simple button. The button needs to have some useful features, such as being able to choose when the button would fire (such as "buttonUp" or "buttonDown"), and providing states such as "pressed", "entered", and "selected". Also graphically, we would like to support different kinds of highlights and appearances based on the state of the button, and a flag that totally disables any highlighting.

A button needs to handle user events and interpret them accordingly. As described in section 4.3, value streams are created and bound to the event type names in the box:

```
this.buttonDown = this.streamOf(undefined)
this.buttonUp = this.streamof(undefined)
this.pointerLeave = this.streamof(undefined)
this.pointerEnter = this.streamof(undefined)
this.motionQuery = this.streamof(undefined)
```

Alternatively, there is a shorthand notation for this repetition. You can just write the following to get the same effects as above:

```
this.listen(["buttonDown", "buttonUp",
             "pointerLeave", "pointerEnter",
             "motionQuery"])
```

There are some states for the button. `pressed` and `entered` are defined in terms of `mergeE()` so do not have any initial value, but if an initial value is supplied using `startsWith()` they behave as behaviors in the FRP sense:

```
this.pressed = $$(mergeE(!!@buttonDown, !@buttonUp, !@pointerLeave))
                    .startsWith(false)
this.entered = $$(mergeE(!!@pointerEnter, !@pointerLeave))
                    .startsWith(false)
this.actsWhen = this.streamOf("buttonUp")
this.selected = this.streamof(false)
```

(Recall that `!` is a logical negation operator but in JavaScript style, where non-nil and non-undefined values are treated as true, applying `!` to the values of a stream of raw-events means to convert them to Booleans.)

The `mergeE` for `pressed` effectively updates itself with the latest value among `buttonDown`, `buttonUp` and `pointerLeave`. Regardless of the previous state, when `buttonDown` is the last event delivered (among these three event types) to the button, the `pressed` state becomes true. Similarly, when `buttonUp` or `pointerLeave` is the last event delivered (among these three event types) to the button, the `pressed` state becomes false.

The `entered` state similarly looks at `pointerEnter` and `pointerLeave`.

With these states, one can write the definition of `clicked` stream, which is updated when the user indeed clicks the button:

```
this.clicked = $$(@buttonUp.doE(() -> @pressed'))
```

The above could have been written as follows:

```
this.clicked = $$(@buttonUp && @pressed')
```

But the difference is that `pressed` is not a dependency of `clicked` in the former case while it is in the latter, as the reference to `pressed` is only used in a function literal that is opaque to the compiler. The problem with `pressed` being a dependency is that `clicked` should get a new value only when a `button up` event occurs, and the previous state of `pressed` is looked at only after that. But if the `pressed` state is a dependency, merely pressing the button would change the `clicked` state. The `pressed'` needs the `'` as the semantics of `clicked` is "mouse button is released when the button **was** pressed previously".

With these states and events, we can now write a stream that really makes a button a button; namely, we add the `fire` stream, which updates when the button should trigger some actions. When does a button fire? In usual cases,

clicking (mouse down and then up) is the right gesture, but in some other cases the button should fire as soon as the pointer button is pressed down. To support this, a variable called `actsWhen` is added, and the stream that represents button firing, called `fire`, looks like:

```
this.fire = $$(mergeE(
    @actsWhen == "buttonUp" && @click,
    @actsWhen == "buttonDown" && @buttonDown).mapE((x) ->
        var ev = if @actsWhen == "buttonUp"
                    @buttonUp
                else
                    @buttonDown
if x then {item: this, event: ev} else undefined))
```

As the change in `buttonDown` from true to false (not only false to true) is propagated to the `mapE` sub-stream, the function checks its actual value (`if x ...`), and filters out when the value was false. The actual value is a new object with `item` and `event` field to denote which object fired by what event (See 5.2 for the use of these values).

The above 12 lines or so is just enough to convert a behavior-less box to a button.

One of the merits of this style is that specifying changes in appearance (such as highlighting/unhighlighting the button) is totally separated from the logical specification. The central stream to control the graphics appearance is called `looks`, whose value is a dictionary of `fill` and `borderFill` for the button. There is another stream called `changeFill` to cause the actual change of appearance in the associated box.

```
this.highlightEnabled = this.streamOf(true)
this.looks = $$(mergeE(
    @entered.mapE((xx) ->
        if @highlightEnabled then xx else undefined).mapE((x) ->
            if x then this.enteredLooks() else this.defaultLooks()),
    @pressed.mapE((xx) -> if @highlightEnabled then xx else undefined).mapE((x) ->
        if x then this.pressedLooks() else this.defaultLooks()),
    @selected.mapE((xx) ->
        if xx then this.selectedLooks() else this.unselectedLooks())))

this.changeFill = $$(@looks.mapE((fills) ->
    if fills.fill
        this.fill(fills.fill)
    if fills.borderFill
        this.borderFill(fills.borderFill)))
```

Notice that when `highlightEnabled` is false, the value of `looks` is `undefined` thus the downstream `changeFill` does not get updated.

Figure 10: A simple menu.

5.2 Menus

In the KSWorld, menus are lists of buttons. There is a method that sets up a box to be a menu. The method has two parts. The first part is written procedurally to create a number of buttons from a given spec (i.e., a loop iterates over the spec and creates buttons). The list of buttons is stored in a field called `items`. The second part sets up the streams to bring the menu to life.

```
this.stayUp = ... // a constant stream of true or false
                  // to denote that menu should stay up
                  // on screen when the user triggers
                  // one item or clicks away
this.items = ...  // the ordered collection of buttons
this.menuActions = ... // the dictionary of buttons to selectors
this.itemSelected = $$(switchColE(@items, "fire"))
this.doAction = $$(@itemSelected.mapE((ev) ->
var menuAction = this.menuActions[ev.item]
@actionTarget.send(menuAction[0], [ev])))
this.dismiss = $$(((@itemSelected || @buttonUp) && !@stayUp)
                  .mapE((x) ->
              if x then this._delete() else undefined))

this.listChangeRequest = this.streamOf(undefined)
```

Again, the `items` field stores the collection of button objects (i.e., boxes that have the streams explained in section 5.1). `itemSelected` uses `switchColE` to detect when any button in `items` fires. When `itemSelected` occurs, the `doAction` stream is triggered and the function is invoked. (The `fire` stream of the button contains an object with a field called `item` to hold onto the button itself; thus the action can be looked up in `menuActions` and can be invoked.) The `dismiss` stream has `itemSelected` and `buttonUp` (and `stayUp`) as its dependencies. If an item is selected or mouse button goes up (could be outside of the menu if `focus` is set to the menu) *and* `stayUp` is false, the menu will be deleted from the screen.

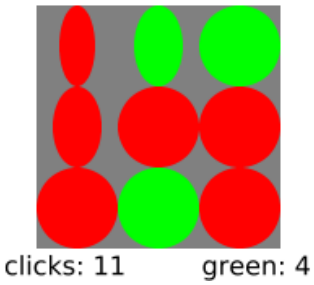

Figure 11: a simple game done in KScript

Again, this shows that connecting various objects can be done with a handful of streams.

5.3 So Saru Game

There may be a generic name for a similar game, but we implemented a simple "coin-flipping" game called "So Saru Game" [16] in KScript as an experiment. The idea of the game is to put coins in 3×3 grid with head and tail randomly. After that, a click on a coin flips the clicked coin as well as the ones horizontally or vertically adjacent. The goal is to make all coins show their faces. (See figure 11. The top-left coin was clicked and the one below and the one to its right as well as itself are being flipped.) What makes this simple game fun is the flip animation. Such animation is a good exercise for a GUI framework such as KSWorld.

5.4 Bridge Simulation

For a larger application, we've created an homage to Ivan Sutherland Sketchpad [20] in the predecessor of our system and shown at the ACM A.M. Turing Centenary Celebration Event [1]. The preceding version of KSWorld did not have `Box` objects be KSObjects but fixed field objects, and the evaluation of streams are individually done for each `Box`, but otherwise it is similar to the system as of writing.

The logic of simulation of connected springs was potentially a good target to write in the dataflow manner with the `'` to compute the simultaneous equations, but in the actual demo shown we instead wrote the core with a traditional loop and copied buffer. However, all interactions including buttons, scene transitions, and connectors are written in the dataflow language to connect events and actions. This was promising as this kind of demo was easily within the boundary of what KSWorld can do (even in its very early stage of development.)

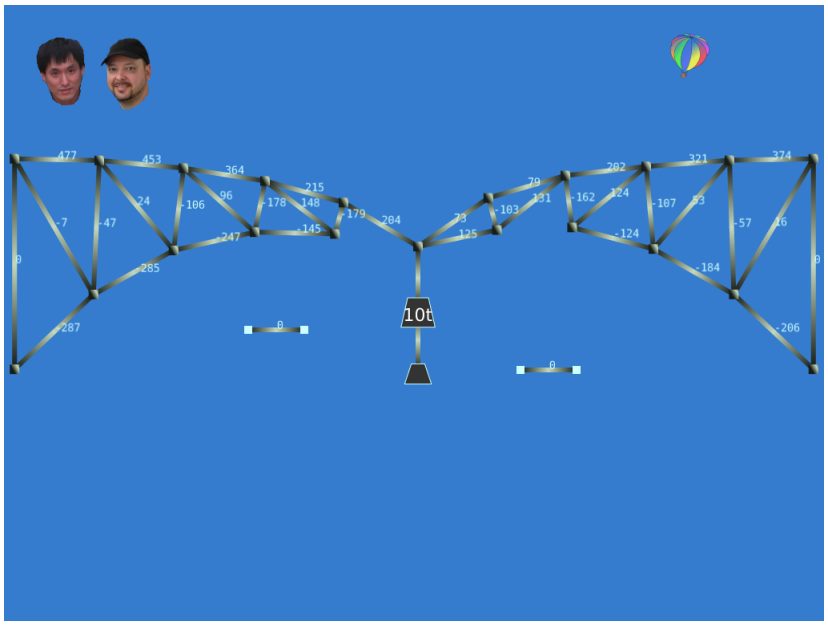

Figure 12: Simulation of a Sketchpad-like bridge with modernized graphics

6 Implementation

In this section, the translation of the KScript into a host language is described.

6.1 Hosting Environment: Squeak

As of this writing, the KSWorld is hosted on top of Squeak Smalltalk [10]. A special Morphic widget called `KSMorph` is created under RectangleMorph. The morph just pretends to be a low-level interface to hardware; the user events delivered to the morph are converted to the events in KSWorld. The morph holds a single bitmap as if it is a frame buffer, and the rendering is done onto the buffer by the Gezira engine, and then Morphic dislays the bitmap. (When OpenGL is enabled, the framebuffer is even bypassed).

The compiler of the KScript language is implemented in OMeta2/Squeak [22]. The compiler parses the input text and after several translation steps (up to eight steps; however half of them are single-function small visitors), it generates Smalltalk code.

The translator is written in OMeta with simple semantic actions, and the KScript only needs sequenceable collection and dictionary as its classes/data structures. The KSWorld GUI framework does not require much features from the OS (again, the graphics engine is written by our group). We strongly hope that we can get off the Squeak scaffolding and make a truly self-sustainable system.

6.2 The Translation Strategy: the First-order Approximation

The first order approximation of the strategy for translating a stream definition in a $$(...) quotation is to expand each sub-expression in it into creation of an object called `EventStream`. A stream definition that looks like following:

```
this.dividedTimer = $$(@timer / 100)
```

would be converted to the code in the base KScript language:

```
this.dividedTimer = new EventStream(["timer"], () -> this.timer.currentValue() / 100)
```

where the first argument for the EventStream constructor is a list of names that this stream depends on, and the second argument is the function to be evaluated when a dependency has changed.

When the expression is more than just one operation, such as following:

```
this.something = $$((@s1 + 100) * @s2)
```

would be converted to:

```
this.tmp1 = new EventStream(["s1"],
                            () -> this.s1.currentValue() + 100)
this.something = new EventStream(["tmp1", "s2"],
                                 () -> this.tmp1.currentValue()
                                       * this.s2.currentValue())


this.s1 = $$( ... )
this.s2 = $$( ... )
```

Namely, the sub-expressions are lifted to the top level and given temporary names.

The evaluator enumerates all streams bound in `this`, sort them based on the dependencies among them. Then it iterates over the sorted list and evaluates the ones that have out standing input.

6.3 The Translation Strategy: the Second-order Approximation

There are a few problems with the strategy described above. One problem is that the temporary names are littering the namespace of the owning object. Since the sub expression is owned by the top-level EventStream, one can convert the above to:

```
this.something = new EventStream(["tmp1", "s2"],
                         () -> this.something.tmp1.currentValue()
                               * this.s2.currentValue())
this.something.tmp1 = new EventStream(["s1"],
                                      () -> this.s1.currentValue() * 100)
this.s1 = $$( ... )
this.s2 = $$( ... )
```

7 Details and Discussions

The previous sections describe the current status without telling a lie, but there are always more details in this kind of large system.

7.1 Language Syntax

The CoffeeScript-like syntax we adapted is a small cleanup from commonly used JavaScript, and also the variable length dictionary-like object is a good match to be used as a namespace. There could be a lot of other choices. For example, Smalltalk-like keyword syntax for messages has better readability, and a Self-like implicit receiver would help with representation independent programming. Another problem in the syntax is that the syntax of the embedded dataflow language is too similar to the base language, even though their results differ widely.

7.2 Assignments

In the current formulation, you can "set" a new value into streams. However, this change often does not get picked up by its dependents until the next evaluation cycle. For example, one can write:

```
this.func = () -> this.transformation.set(...)
this.myAction = $$(@buttonDown.doE() -> this.func())
```

The problem is that the `myAction` stream itself cannot tell that it would update the `transformation` stream; in other words, such update to `transformation` occurs under the radar of the dataflow evaluator and becomes the source of glitches.

One clean solution would be along the lines of the Kaleidoscope language families [13], where such an assignment is a promise of future modification and does not take effect until the next evaluation cycle. But as described later in section 7.4, pushing things into the future has cascading effects and introduces the need for waiting until the system "settles down". With a hybrid system like KSWorld where some actions are not functional (such as a function that makes a beep from the computer speaker), we need to ensure the right timing of evaluation.

7.3 Virtual Fields

As described in 4.1, there are fundamental streams such as `transformation` and `shape` as well as derived ones such as `extent` and `bounds`. (Recall that `bounds` is the bounding rectangle of the box in the parent coordinate system, and it depends on the `transformation` and `extent`, which in turn depends on the drawable description in `shape`.)

Furthermore, as described in 4.2, one could imagine that properties such as `center` are derived from `bounds`. In analogy, these derived properties are similar to Tweak's virtual fields [19].

One possible formulation of these geometric properties is to make all of them be streams. `bounds` is a stream that depends on `transformation` and `extent`, and `center` is a stream that depends on `bounds`. This is nice because the user can just write a stream that depends on `center`, or `bottomRight` of a box:

```
this.averager = $$(((@aBox.center + @bBox.center) / 2.0)
                    .mapE((c) -> ...))
```

We tried this approach but ran into a problem.

The problem was that the values in these streams are not always in sync (there are "glitches" in the FRP terminology). The reason was that the update to `transformation`, for example, can be done by procedural code. In that case, resolving the value of `bounds` and `center` only happens in the next evaluation cycle, but there could be a stream that depends on `transformation` and `center`; the values it sees are inconsistent and would lead into more inconsistency.

This problem was further aggravated when we want to have the bi-directional constraints idea incorporated. If you can depend on the `center` of a box, why can't you expect that specifying a box's center does the Right Thing? If so, one could write something like:

```
this.center := $$((@aBox.center + @bBox.center) / 2.0)
```

and the different style of assignment (`:=`) would trigger the setter expression for `center`, which would result in signaling the fundamental stream `transformation` to be updated from the change in `center`.

The bi-directionality is desirable in many situations, but if we take the dataflow model, which is inherently uni-directional, we cannot achieve it. If no rotation or scaling is involved, the relationship between these properties is linear so a linear constraint solver such as Cassowary [6] would work. However, we would like to have non-linearity because of the full transformation. In the future, we would like to base our system on stronger (non-linear) solvers to provide usable abstraction for manipulating boxes.

In section 4.2, we discussed the mutator in differential form (e.g., `translateBy()`, `rotateBy()`, etc.) These could have been written in the dataflow language extension, as it is like taking the old value of (for example) `rotation` and incrementing it with the given argument to make the new value. But this entails overriding the default implementation of `transformation`, and again makes the system inflexible.

7.4 Different Evaluation Strategy

Roughly dividing various implementations of FRP systems, there are two ways of evaluating the streams; one is push-based and another is pull-based. Currently, KSWorld's strategy could be described as "pull-based but all streams are looked

at". The main benefit of this strategy is that one does not have to register, or keep the pointer to, any stream in any other streams. Such pointers would make garbage collection and serializing streams harder, and more brittle to changes.

However, if we don't stick with FRP, there is a third possibility. In this third possibility, the system would not even try to sort, but any stream is always looking at its dependencies' old values to compute its current value. (As if all variables in the formula always have ' attached.)

In the pure functional setting, this approach has big advantages. One issue, though, is that the system goes through states that are not consistent with the "contract" the user gives to the system. For example, if there is a set of KScript streams defined as follows:

```
this.a1 = this.streamOf(1)
this.a2 = $$(@a1' + 1)
this.a3 = $$(@a2' + 1)
this.a4 = $$(@a3' + 1)
```

and the user updates the value of a1 to be 2 at time 1, then the propagation of this change is as shown in table 1.

time	a1	a2	a3	a4
0	1	2	3	4
1	2	2	3	4
2	2	3	3	4
3	2	3	4	4
4	2	3	4	5
5	(2)	(3)	(4)	(5)

Table 1: The progress of computation on a set of streams

The user's expectation is that the "contracts" such as `a2` should be *always* `a1 + 1` and `a4` should *always* `a1 + 3` are maintained, at least when he observes the system. But in this evaluation scheme the length of the causal chain affects the number of cycles needed to propagate the change. Displaying the results on screen, or serializing the data and sending them over a network, should not expose such intermediate inconsistent states, so the evaluator has to be able to detect when the system settles into a stable state. It is the same problem with assignment taking effect in the next evaluation cycle.

Under these observations, we are still operating with the idea of sorting streams but we hope that there are better ways to handle it.

For example, we should think of FRP as just a uni-directional, equality-only simple constraint solver. Given that more sophisticated (constraint) solvers exist, we would like to integrate such a solver. Again, the Kaleidoscope family of languages is worth revisiting, as the set of constraints can be solved "at once". We would need a good separation of pure computation and stateful action (such as beeping the speaker) in such a framework.