Coherent Reaction

Jonathan Edwards
MIT Computer Science and Artificial Intelligence Lab
edwards@csail.mit.edu

Abstract

Side effects are both the essence and bane of imperative programming. The programmer must carefully coordinate actions to manage their side effects upon each other. Such coordination is complex, error-prone, and fragile. *Coherent reaction* is a new model of *change-driven* computation that coordinates effects automatically. State changes trigger events called *reactions* that in turn change other states. A *coherent* execution order is one in which each reaction executes before any others that are affected by its changes. A coherent order is discovered iteratively by detecting incoherencies as they occur and backtracking their effects. Unlike alternative solutions, much of the power of imperative programming is retained, as is the common sense notion of mutable state. Automatically coordinating actions lets the programmer express *what* to do, not *when* to do it.

Coherent reactions are embodied in the Coherence language, which is specialized for interactive applications like those common on the desktop and web. The fundamental building block of Coherence is the dynamically typed mutable tree. The fundamental abstraction mechanism is the *virtual tree*, whose value is lazily computed, and whose behavior is generated by coherent reactions.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.3 [*Programming Techniques*]: Concurrent Programming; F.1.2 [*Computation by Abstract Devices*]: Modes of Computation—Interactive and reactive computation

General Terms Languages

Keywords interactive systems, reactive systems, synchronous reactive programming, functional reactive programming, bidirectional functions, trees

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

1. Introduction

> *I see no reasonable solution that would allow a paper presenting a radically new way of working to be accepted, unless that way of working were proven better, at least in a small domain.* – Mark Wegman, What it's like to be a POPL referee [28]

This paper presents a new kind of programming language and illustrates its benefits in the domain of interactive applications (such as word processors or commercial web sites). The fundamental problem being addressed is that of *side effects*, specifically the difficulties of **coordinating side effects**.

Coordinating side effects is the crux of imperative programming, the style of all mainstream languages. Imperative programming gives the power to change anything anytime, but also imposes the responsibility to deal with the consequences. It is the programmer's responsibility to order all actions so that their side effects upon each other are correct. Yet it is not always clear exactly how actions affect each other, nor how those interdependencies might change in the future.

Coordinating side effects is a major problem for interactive applications, for two reasons. Firstly, interaction *is* a side effect. The whole purpose of user input is to change the persistent state of the application. The issue can not be side-stepped. Secondly, the size and complexity of modern applications demands a modular architecture, such as Model View Controller (MVC) [26] and its descendants. But coordination of side effects inherently cross-cuts modules, leading to much complexity and fragility.

A common example is that of a model constraint that ensures multiple fields have compatible values. If a view event, say submitting a form, changes two of those fields, it is necessary that they both change before the constraint is checked. Otherwise the constraint might falsely report an error. There are many common workarounds for this problem, none of them entirely satisfactory.

In the MVC architecture it falls to the controller to coordinate change. One approach to the example problem is to defer checking the constraint until after all relevant changes have been made. This erodes modularity, for now the model must publish all its constraints and specify what fields they

depend upon, limiting the freedom to changes such internals. Even still it may not be obvious to the Controller what implicitly called methods may changes those fields, so it can not be sure when to call the check. It could defer all checks till the very end. But that presumes that the code is itself never called by other code, again eroding modularity. The difficulty of modularizing MVC controllers has been discussed by others. [3, 17, 24]

Another approach, no more satisfactory, is to have the model publish special methods that bundle the changes to all constraint-related fields into a single call. Once again this defeats modularity, for the model is exposing its internal semantics, limiting the freedom to change them. Worse, the controller is given the impossible task of accumulating all changes to the relevant fields, made anywhere in the code it calls, so that it can change them atomically.

Modern application frameworks employ an event-driven publish/subscribe model to respond to input more modlarly. Event handlers can subscribe to be called back whenever an event is published. The subscribers and publishers need not know of each other's existence. This approach eliminates many hard-wired interdependencies that obstruct modularity, but does not solve the example problem. The constraint can not subscribe to changes on the involved fields, for it will be triggered as soon as the first one changes. One response is to queue up the constraint checks to be executed in a separate phase following all the model change events. The popular web framework JavaServer Faces [4] defines ten different phases.

Phasing is an ad hoc solution that works only for preconceived classes of coordination problems. Unfortunately event-driven programming can create more coordination problems than it solves. The precise order of interrelated event firings is often undocumented, and so context-dependent that it can defy documentation.[1] You don't know when you will be called back by your subscriptions, what callbacks have already been called, what callbacks will be subsequently called, and what callbacks will be triggered implicitly within your callback. Coordinating changes to communal state amidst this chaos can be baffling, and is far from modular. The colloquial description is *Callback Hell*.

An analysis [21] of Adobe's desktop applications indicated that event handling logic comprised a third of the code and contained half of the reported bugs.

The difficulties of event coordination are just one of the more painful symptoms of the disease of *unconstrained global side effects*. It has long been observed that global side effects destroy both referential transparency [19] and behavioral composition [16]. Unfortunately, attempts to banish side effects from programming languages have required significant compromises, as discussed in the Related Work section.

The **primary contribution** of this paper is *coherent reaction*, a new model of change-driven computation that constrains and coordinates side effects automatically. The **key idea** is to find an ordering of all events (called *reactions*) that is *coherent*, meaning that each reaction is executed before all others that it has any side effects upon. Coherent ordering is undecidable in general. It can be found with a dynamic search that detects incoherencies (side effects on previously executed reactions) as they occur. All the effects of a prematurely executed reaction are rolled back, as in a database transaction, and it is reexecuted later. From the programmer's point of view, coordination becomes automatic. The programmer can concentrate on saying *what* to do, not *when* to do it. Coherent reaction is discussed in more detail in the next section.

The Coherence programming language uses coherent reactions to build interactive applications. The fundamental building block of the language is the dynamically typed mutable tree. The **key idea** is that abstraction is provided by *virtual trees*, whose values are lazily computed, and whose behaviors are generated by coherent reactions. Further details can be found in the full version [11] of this paper.

2. Coherent Reaction

This section explains coherent reaction in the simple setting of a Read Eval Print Loop (REPL). Programmer input is prefixed with a >, the printed value of inputs with a =, and program output with a <. Printed values will be omitted when they do not further the discussion.

```
1  > task1: {
2      name: "task1",
3      start: 1,
4      length: 2,
5      end = Sum(start, length)}
6  = {name: "task1", start: 1, length: 2, end: 3}
7  > task1.start := 2
8  > task1
9  = {name: "task1", start: 2, length: 2, end: 4}
```

Lines 1–5 define the variable task1 to be a structure containing the fields within the curly braces. This structure is meant to represent a task in some planning application, and has a starting time and length defined in the fields start and length. For simplicity these fields are given plain numeric values rather a special time datatype. Variables and fields are dynamically typed.

The field end is defined on line 5 as the total of the start and length fields using the Sum function. (Functions are capitalized by convention. Traditional infix mathematical notation can be supported, but will not be used in this paper.) The end field is said to be *derived*, indicated by defining it with an equals sign instead of a colon, followed by an expression to calculate the value. The value of task1 is printed on 6, with end correctly computed.

[1] For example, when the mouse moves from one control to another, does the mouseLeave event fire on the first before the mouseEnter event fires on the second? Does your GUI framework document that this order is guaranteed? The order is seemingly random in one popular framework.

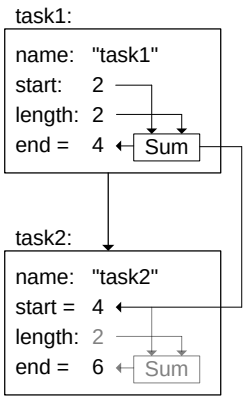

Figure 1. Arrows denote derivation, faded elements are inherited.

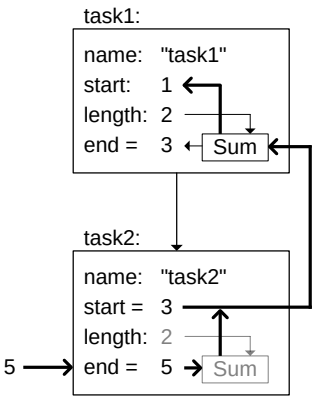

Figure 2. Reaction flow. Bold arrows show the flow. Values are post-states.

The derivation expression of end is recalculated every time the field is accessed (although the implementation may cache the last calculated value and reuse it if still valid). The persistence of derivation is demonstrated on line 7, where an assignment statement changes the value of the start field. (Assignment statements use the := symbol instead of a colon or equals sign.) The effect of the assignment is shown by referencing task1 on line 8, whose value is output on line 9, where the derived value of end has been recomputed.

Derivation is a fundamental concept in Coherence. A derivation is computed lazily upon need, and as will be seen is guaranteed to have no side-effects, so it is like a well-behaved getter method in OO languages. A derivation expression is also like a formula in a spreadsheet cell: it is attached to the field and continually links the field's value to that of other fields. The following example shows more ways that derivation is used.

```
10  > task2: task1(name: "task2", start = task1.end)
11  = {name: "task2", start: 4, length: 2, end: 6}
```

Line 10 derives the variable task2 as an *instance* of task1, meaning that it is a copy with some differences. The differences are specified inside the parentheses: a new name is assigned, and the start field is derived from the end field of task1. Figure 1 diagrams this example. The length field was not overridden, and is inherited from the prototype, as shown by its value output on line 11. Any subsequent changes to task1.length will be reflected in task2.length. However since task2.name has been overridden, changes to task1.name will not affect it. Derivation functions are inherited and overridden in the same manner. Instantiation behaves as in prototypical languages [25]. Functions are also treated as prototypes and their calls as instances (justifying the use of the same syntax for calling and instantiation). Materializing execution in this way has large ramifications on the design of the language, including the interpretation of names and the status of source text [9, 10], but those issues are beyond the scope of this paper.

2.1 Reaction

Derivation is bidirectional: changes to derived variables can propagate back into changes to the variables they were derived from. This process is called *reaction*, and is used to handle external input. A Coherence system makes certain structures visible to certain external interfaces (the programmer's REPL can see everything). All input takes the form of changes to such visible fields, which react by changing internal fields, which in turn can react and so on. Multiple input changes can be submitted in a batch, and the entire cascade of reactions is processed in a transaction that commits them atomically or not at all. Output consists of reading visible fields, which are rederived if necessary from the latest changed state. The following example illustrates.

```
12  > task2.end := 5
13  > task2
14  = {name: "task2", start: 3, length: 2, end: 5}
15  > task1
16  = {name: "task1", start: 1, length: 2, end: 3}
```

On line 12 the task2.end field is assigned the value 5, and the results are shown on the following four lines and diagrammed in Figure 2. Because task2.end is a derived field its derivation function reacts to the change. Every function reacts in some way, if only to declare an error. The Sum function's reaction is to adjust its first argument by the same amount as the result, so that the result is still the sum of the arguments.[2] Thus a change to the end of a task will adjust its start to maintain the same length: task2.start is changed to 3. Since task2.start is derived from task1.end, the reaction propagates to the latter field, and in turn causes task1.start to be set to 1. The task1.start field is not derived, so the chain reaction *grounds out* at that point, leaving the field changed. If you don't like the built-in reaction of a function you override it with your own custom reaction, as follows.

[2] The second argument could be adjusted instead. A function is expected to document such choices.

```
17  > task1: {
18      name: "task1",
19      start: 1,
20      length: 2,
21      end = Sum(start, length)
22        => {start := Difference(end', length')}}
23  = {name: "start1", start: 1, length: 2, end: 3}
```

Here task1 has been redefined to include a custom reaction for the end field on line 22. The symbol => specifies the reaction for a field, in counterpoint to the use of = for the derivation, as reaction is opposite to derivation. The reaction is specified as a set of statements that execute when the field is changed. Note that while these statements may look like those in a conventional imperative language, they behave quite differently. For one thing, they execute in parallel. Section 2.3 will explain further.

The reaction specified above duplicates the built-in reaction of the Sum function. The changed value of the end field is referenced as end', adopting the convention of specification languages that primed variables refer to the post-state. Non-primed references in reactions always refer to the pre-state prior to all changes in the input transaction. The post-state of end has the post-state of length subtracted from it to compute the post-state of start. The post-state of length is used rather than its pre-state because it could have changed too, discussed further below.

2.2 Actions

Reactions can make arbitrary changes that need not be the inverse of the derivation they are paired with. An example of this is numeric formating, which leniently accepts strings it doesn't produce, effectively normalizing them. An extreme case of asymmetry is an *action*, which is a derivation that does nothing at all and is used only for the effects of its reaction. Here is a "Hello world" action.

```
24  > Hello: Action{do=>{
25      consoleQ << "Hello world"}}
26  > Hello()
27  < Hello world
```

The Hello action is defined on line 24 with the syntax Action{do=>...}. This indicates that a *variant* of the prototype Action is derived, incorporating a reaction for the do field. A variant is like an instance, except that it is allowed to make arbitrary internal changes, whereas instances are limited to changing only certain public aspects like the input arguments of a function. Curly braces without a leading prototype, like those used to create the task1 structure in line 17, are actually creating a variant of null, an empty structure.

Actions are triggered by making an arbitrary change to their do field (conventionally assigning it null), which has the sole effect of triggering the reaction defined on it. A statement consisting of only a function call will trigger its action by changing its do field. The Hello action is triggered in this way on line 26. By encoding actions as the reactions of do fields we establish the principle that all behavior is in reaction to a change of state, which is essential to the semantics described below.

The body of the action on line 25 outputs to the console with the syntax consoleQ<<"Hello world". The << symbol denotes an *insertion* statement. It creates a new element within consoleQ and assigns its value to be the string "Hello world". If the input transaction commits, any elements inserted into the consoleQ will be printed and then removed from the queue. Driving console output from a queue preserves the principle that all behavior is in reaction to a change of state.

2.3 Coherent execution

Enough preliminaries are now in place to explain the semantics of coherent reactions. Say that inside some action we need to change a task's end and length, as in the following code snippet.

```
28  TaskAction: Action{task, do=>{
29    ...
30    task.end := e,
31    task.length := d}}
```

The question is, what is the value of the task's start field afterwards? One might expect it to be $e - d$. That would be wrong if this code were executed in an OO language, where the reaction of end would be encoded into its set method. The set method would use the value of length at the time it was called to calculate start. But length is set after the call, so the value of start will actually be $e - oldLength$ and the value of end recalculated by its get method will not be e as expected but $e - oldLength + d$.

Obviously it is necessary to set length before end to get the correct result. But in practice such issues are often far from obvious. The side-effects of methods (especially those caused by deeply nested method calls) are often undocumented and subject to change. For example if task were refactored so that length was instead derived from the difference of start and end, then any code like ours depending on the ordering of the side-effects would break. This example is indicative of the fundamental quandary of imperative programming: it is up to the programmer to orchestrate the exact order in which all events takes place, yet the programmer often lacks the omniscience and clairvoyance required to do so perfectly. The result is much complexity and fragility.

Coherence avoids these problems by automatically determining the correct execution order of all events. In the above example, the reaction on end will be automatically executed after the assignments to end and length. A correct execution order is called *coherent*, defined as an order in which every reaction executes before any others that it affects. A reaction affects another in only one way: if it writes (assigns to) a location whose post-state is read by the other.

Finding a coherent order may seem at first to be a straightforward problem of constraint satisfaction. We form a graph of reactions whose edges are such effects. A coherent order is a topological sort of this graph. The problem is that forming this graph is undecidable. Reactions can use pointers:

they are free to do arbitrary computations to compute the locations which they read and write. For example, TaskAction might take some user input as a key with which to search all tasks with a spelling-similarity algorithm, and then modify the found task. Allowing arbitrary computation of locations makes the effect graph undecidable in general. Coherence is not a problem of constraint *satisfaction* — it is a problem of constraint *discovery*. Previously there have been two alternative solutions: reduce the expressive power of the language so that constraint discovery becomes decidable (as in state machines and dataflow languages), or leave it to the programmer to deal with.

This paper introduces a new technique that dynamically discovers effects between reactions and finds a coherent execution order. Every reaction is run in a *micro-transaction* that tracks both its writes and post-state reads. Reactions are initially executed in an arbitrary order. *Incoherencies* are detected as they occur: whenever a reaction writes a location whose post-state was read by a previously executed reaction. In that case the previous reaction's micro-transaction is aborted and it is run again later. The abort cascades through all other reactions that were transitively affected. This algorithm is essentially an iterative search with backtracking, using micro-aborts to do the backtracking. If there are no errors a coherent execution order will found and the whole input transaction is committed.

Cyclic effects are an error: a reaction can not transitively affect itself. Errors are handled tentatively because they might be later rolled back — errors that remain at the end cause an abort of the whole input transaction. The search for a coherent ordering converges because reactions are deterministic (randomness is simulated as a fixed input). It will terminate so long as the reactions themselves terminate, as only a finite number of reactions can be triggered.

2.4 The price of coherence

Clearly a naive implementation of coherence will be slower than hand-written coordination logic in an imperative language. But at this point worrying about performance optimization would be both premature and misguided. The history of VM's shows that clever implementation techniques can yield large speedups. There is a large body of prior research that could be exploited, from static analysis to feedback-directed optimization. Coherent code reveals inherent parallelism that might be exploited by multicore processors. Annotations could partially instruct how to order reactions (but still be checked for coherence, which is easier than solving for it). In any case the major problem of interactive applications is not CPU performance but programmer performance — the difficulty of designing, building, and maintaining them.

Coherence imposes certain constraints on reactions:

1. A field can change at most once per input transaction. Multiple reactions can change the same field, but only to the same value. This situation might occur in the above example if the code snippet also assigned the start field. That would be OK so long as the value agreed with what the reaction computed it should be, which would effectively become an assertion: if the values disagreed an error would abort the input transaction.
2. All reactions can see the entire global pre-state. Each can see the pending post-state of the field it is attached to, and decides how to propagate those changes to other fields. Each can also see the pending post-state of other fields. But in no case can a reaction see the consequences of any changes it makes, because that would create a causal loop whereby it depends upon itself. Causality violation is punished by aborting the transaction.
3. A consequence of the above property is that all of the assignment statements inside a reaction execute as if in parallel. Causal ordering only occurs between different reactions.

This paper suggests that much of the manual sequencing of actions that is the crux of imperative programming is an accidental complexity [2], and that coherent execution can handle it automatically, at least in the domain of interactive applications. But there are cases when sequential execution is truly essential. For such cases, Coherence offers an encapsulated form of imperative programming called *progression*.

2.5 Progression

Say that we want to execute the previous TaskAction on a task, but also want to ensure that whatever it does, the task's length ends up no more than 10. We could do that by creating an alternate version of TaskAction that maximized the length before assigning it. But it is simpler to just execute TaskAction and then cap the length if it is too large. However reactions only get a single shot to change each field, and can not see the consequences of their own actions. Instead we can use a progression:

```
32  BoundedAction: Action{task, do=>{
33    prog (task) [
34      TaskAction(task);
35      if (Gt(task.length, 10)) then
36        {task.length := 10}]}}
```

The **prog** statement on line 33 takes a parenthesized list of one or more *versioned* variables, which here is just task. That is followed by square brackets containing a sequence of statements separated by semicolons. The statements can be read somewhat imperatively: the statement on line 34 executes TaskAction on task, and then the **if** statement on the following line checks the resulting length value and sets it to 10 if it is greater. What actually happens is that a separate version of task is made for each statement. Each statement changes its version, which then becomes the pre-state of the next version.

An example reaction flow for BoundedAction(task2) is diagrammed in Figure 3. The first version of task2 is modified by

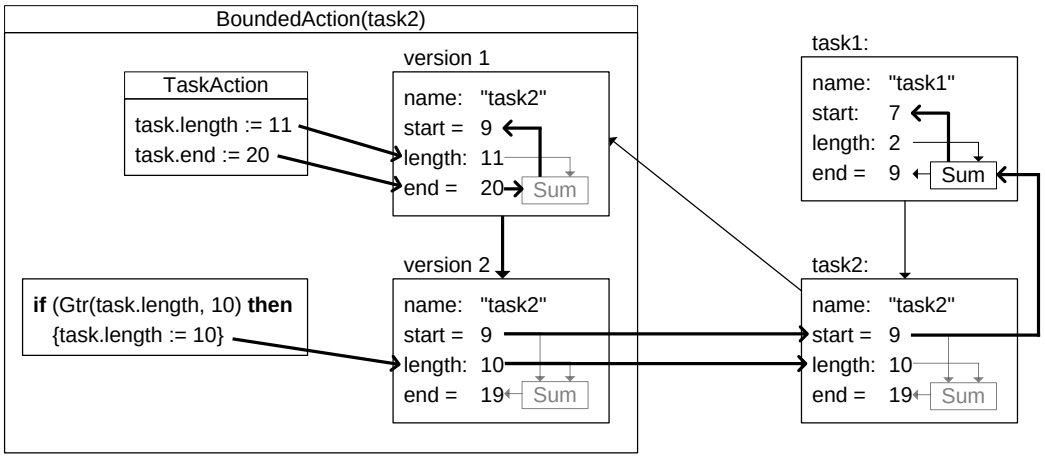

Figure 3. Progression reaction flow. All values are post-states.

TaskAction, creating the second version which is modified by the **if** statement. The changes made in the first version are encapsulated. The change to length gets overridden in the second version, and the change to end is discarded because it is consumed by the Sum reaction. The Sum reaction's change to start gets inherited into the second version. The accumulated changes to start and length in the second version are exported out of BoundedAction. The exported change to task2.start then propagates into task1.end. Note that while internal reactions like Sum execute in each version, any external reactions like the link to task1 only execute at the very end.

Progressions are encapsulated: the internal unfolding of events is isolated from the outside. External changes are visible only at the beginning, and internal changes become visible externally only by persisting till the end.

Progression depends on the fact that Coherence can make incremental versions of entire structures like a task. As discussed in the full version [11] of this paper, the state of a Coherence program is a tree. Progressions can version any subtree, such as collections of structures, or even the entire state of the system. In the latter case, progression becomes a simulation of imperative programming, capable of making arbitrary global changes in each version, and constrained only from doing external I/O. This simulation also reproduces the usual pitfalls of imperative programming. Progression is an improvement over imperative programming only to the extent that it is quarantined within localized regions of state, and used as a special case within a larger coherent realm.

Progressions also support looping with a for statement. The whatif statement is a *hypothetical* progression with no effect, executed only to extract values produced along the way. Hypotheticals function like normal progressions, except that all emitted changes are silently discarded. Values produced within a hypothetical can be accessed from its calling context. Hypotheticals turn imperative code into pure functions, and can thus be used inside derivations. Hypothetical progressions on the global state can be used for scripting behavioral tests, running the entire system in an alternate timeline.

2.6 Coherence as a model of computation

Derivation and reaction are quite different, yet work well together. To summarize:

1. Interaction is cyclic: input and output alternate.
2. Output is derivation: external interfaces query visible state, which may be derived from internal state.
3. Input is reaction: external interfaces stimulate the system by submitting batches of changes to visible fields, which react by propagating changes to internal fields. Input is transactional: all the changes happen atomically or not at all.
4. Derivation (output) is pure lazy higher-order functional programming. It executes only upon demand, and can not have any side-effects. Derivation is discussed further in the full version [11] of this paper.
5. Reaction (input) is coherent. A set of input changes cascades through reactions until they all ground out into state changes or errors. Reactions are automatically ordered so that each executes before any others that it affects. Reactions that transitively affect themselves are an error. Errors abort the entire input transaction.
6. Coherence is dynamic. State can grow and change. Reactions can have arbitrary dynamically computed effects, though they may need to use progressions to do so.
7. Derivation, as pure functional programming, does not naturally handle the state mutation inherent in input. Reaction does not naturally handle output, for that would lead to cyclic effects on inputs. Derivation and reaction need each other.
8. Coherence is the *dual* of laziness. They both remove timing considerations from programming. A lazy function

	Object Orientation	Coherence
Central metaphor	Conversing with messages (language)	Seeing and direct-manipulation (vision and fine motor)
Organization	Autonomously changing objects	Holistically changing tree
Nature of change	Sequential	Parallel
Modularity via	Behavioral substitution	Structural substitution
Interface contract	Temporal patterns of behavior (protocols)	Spatial patterns of values (constraints)
Simulates the other	Structure simulated behaviorally (e.g. Collection protocols)	Behavior simulated structurally (e.g. trigger fields, queues)

Figure 4. OO contrasted with Coherence.

executes before its result is *needed*. A coherent reaction executes before its effect *matters*.

9. It is often said that functional programs let one express *what* should be computed, not *how*. Coherent reactions let one express *what* should be done, not *when*.

These symmetries are pleasing. Derivation and reaction are complementary opposites that fit together like yin and yang to become whole.

3. Related Work

The fundamental issue of this paper, managing side effects, has been researched so extensively that the related work can only be briefly summarized in the space available here. A more detailed consideration is in the full version [11] of this paper.

Many languages that improve state mutation and event handling do so by mandating a static dependency structure, as in dataflow languages [7, 22] and state machines [15]. Such languages do not allow the program to dynamically choose the target of changes: you can not assign through a pointer, or update the result of a query. Likewise for Synchronous Reactive Programming (SRP) [1, 5], which was an inspiration for coherent reaction, and is more general in some ways. But SRP is intended for embedded systems and so limits itself to static state spaces that can be compiled into state machines or gate arrays.

Bidirectional computation is supported in constraint and logic languages, and in Lenses [14]. Such bidirectional computations are symmetric, whereas the asymmetry of derivation and reaction allow arbitrary changes to be expressed.

Trellis [8] is a Python library that appears to have been the first invention of the essential idea of coherent reaction: using transactional rollback to automatically order event dependencies. While Coherence was developed independently of Trellis, the prior work on Reactors [13] was a direct influence. Reactors offer a data-driven model of computation where data is relational and code is logical rules. It could be said that Reactors are to logic programming as Coherence is to functional programming.

Monads [27] simulate imperative programming through higher-order constructions, allowing some parts of the program to remain pure. But all the usual problems of coordinating side effects still exist inside the monadic code. Functional Reactive Programming (FRP) [6, 12, 18] entirely abandons the notion of mutable state. The normal order of cause-and-effect is inverted: effects are defined in terms of all causes that could lead to them. FRP requires that programmers learn a new way of thinking about change. Coherence retains the common sense notions of mutable state and causality, abandoning only the Program Counter.

4. Conclusion

> *Smalltalk's design—and existence—is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages.* – Alan Kay [23]

The conceptual model of Coherence is in a sense opposite to that of Object Oriented languages. As Alan Kay's quote above indicates, the central metaphor of OO is that of messaging: written communication. The central metaphor of Coherence is that of observing a structure and directly manipulating it. These two metaphors map directly onto the two primary mechanisms of the mind: language and vision. Figure 4 contrasts several other language aspects.

The pattern that emerges strikingly matches the division of mental skills into *L-brain* and *R-brain* [20]. From this perspective, OO is verbal, temporal, symbolic, analytical, and logical. In contrast Coherence is visual, spatial, concrete, synthetic, and intuitive. This observation raises a tantalizing possibility: could there be such a thing as an R-brain programming language — one that caters not just to the analytical and logical, but also to the synthetic and intuitive?

Acknowledgments

This paper benefited from discussions with William Cook, Derek Rayside, Daniel Jackson, Sean McDirmid, Jean Yang, Eunsuk Kang, Rishabh Singh, Kuat Yessenov, Aleksandar Milicevic, Frank Krueger, Thomas Lord, and John Zabroski.

References

[1] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: Design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.

[2] F. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE computer*, 20(4), 1987.

[3] S. Burbeck. How to use Model-View-Controller (MVC). Technical report, ParcPlace Systems Inc, 1992.

[4] E. Burns and R. Kitain. JavaServer Faces Specification v1.2. Technical report, Sun Microsystems, 2006.

[5] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, 1987.

[6] G. Cooper and S. Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In *15th European Symposium on Programming, ESOP 2006*, 2006.

[7] J. Dennis. First version of a data flow procedure language. *Lecture Notes In Computer Science; Vol. 19*, 1974.

[8] P. J. Eby. Trellis. June 2009. URL http://peak.telecommunity.com/DevCenter/Trellis.

[9] J. Edwards. Subtext: Uncovering the simplicity of programming. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 505–518. ACM Press, 2005.

[10] J. Edwards. Modular Generation and Customization. Technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory TR-2008-061, October 2008. URL http://hdl.handle.net/1721.1/42895.

[11] J. Edwards. Coherent Reaction. Technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory TR-2009-024, June 2009. URL http://hdl.handle.net/1721.1/45563.

[12] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[13] J. Field, M. Marinescu, and C. Stefansen. Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications. In *Coordination 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*. Springer, 2007.

[14] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3), May 2005.

[15] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3), 1987.

[16] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*. Springer-Verlag New York, Inc., 1985.

[17] G. T. Heineman. An Instance-Oriented Approach to Constructing Product Lines from Layers. Technical report, WPI CS Tech Report 05-06, 2005.

[18] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. *Lecture Notes in Computer Science*, 2638, 2003.

[19] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2), 1989.

[20] A. Hunt. Pragmatic Thinking and Learning: Refactor Your Wetware (Pragmatic Programmers). 2008.

[21] J. Järvi, M. Marcus, S. Parent, J. Freeman, and J. N. Smith. Property models: From incidental algorithms to reusable components. In *Proceedings of the 7th international conference on Generative Programming and Component Engineering*, 2008.

[22] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1), 2004.

[23] A. C. Kay. The early history of smalltalk. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*. ACM, 1993.

[24] S. Mcdirmid and W. Hsieh. Superglue: Component programming with object-oriented signals. In *Proc. of ECOOP*. Springer, 2006.

[25] J. Noble, A. Taivalsaari, and I. Moore. *Prototype-Based Programming: Concepts, Languages and Applications*. Springer, 2001.

[26] K. Pope and S. Krasner. A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1, 1988.

[27] P. Wadler. Monads for functional programming. *Lecture Notes In Computer Science; Vol. 925*, 1995.

[28] M. N. Wegman. What it's like to be a POPL referee; or how to write an extended abstract so that it is more likely to be accepted. *ACM SIGPLAN Notices*, 21(5), 1986.