Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2009-024 June 12, 2009

Coherent Reaction

Jonathan Edwards

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

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

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 [48]

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) [44] 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. [9, 27, 36]

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 [10] 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 [32] 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 [30] and behavioral composition [26]. 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 **secondary contribution** of this paper is the Coherence language, which employs 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. The Coherence language is discussed in section 3.

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

[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.

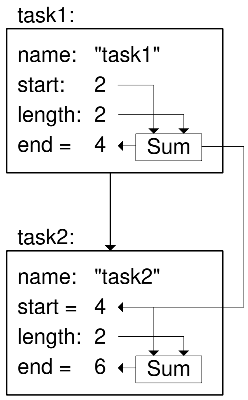

Figure 1. Arrows denote derivation, faded elements are inherited.

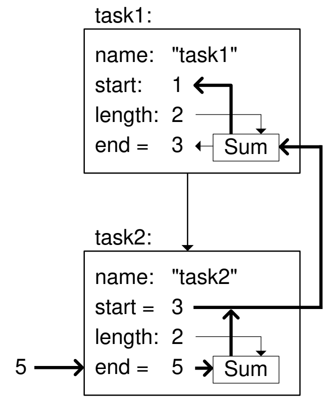

Figure 2. Reaction flow. Bold arrows show the flow. Values are post-states.

calculate the value. The value of `task1` is printed on 6, with `end` correctly computed.

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 [42]. 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 [17, 19], 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

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

changed. If you don't like the built-in reaction of a function you override it with your own custom reaction, as follows.

```
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 [8], 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.

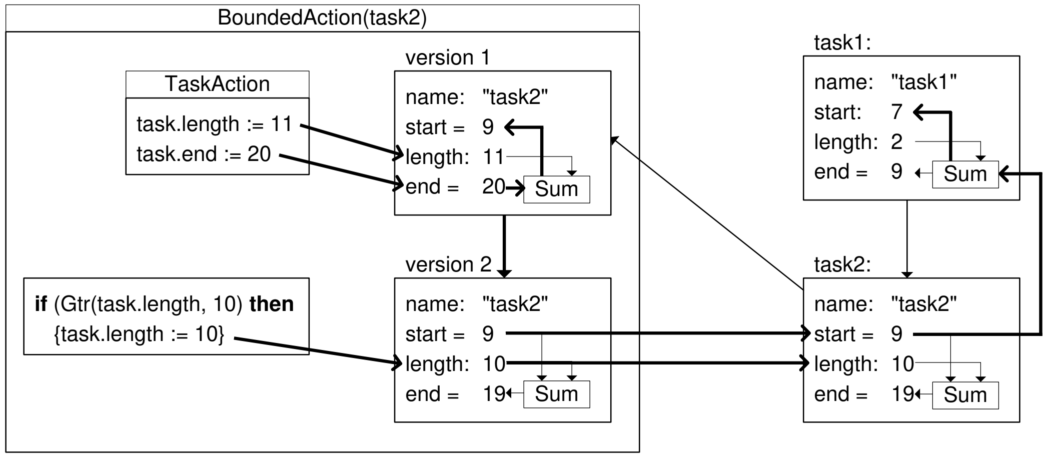

Figure 3. Progression reaction flow. All values are post-states.

An example reaction flow for BoundedAction(task2) is diagrammed in Figure 3. The first version of task2 is modified by 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 later in Section 3, 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 explained more fully in section 3.
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 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. The Coherence Language

This section more fully describes the Coherence programming language, which incorporates the idea of coherent reaction in order to build interactive applications. The fundamental building block of Coherence is the tree. Unlike in functional programming languages, trees are dynamically typed and mutable: pointers can traverse freely through a tree and make changes in situ. The entire state of a Coherence application is kept in a single *state-tree*, including both code and data.

The fundamental abstraction mechanism of the language is the *virtual tree*. Any subtree of the state-tree can be virtualized, meaning that its value is derived by a function. That function lazily computes the contents of the virtual tree top-down as needed. Changes to a virtual tree are handled by the reaction of its derivation function, or a programmer-supplied reaction. Virtual trees look and feel like normal tree data structures, except that their look (their contents) is computed on the fly by their derivation, and their feel (their response to changes) is generated by their reaction. Modularity in Coherence is achieved by substituting trees with similar structure for one another. They can transparently differ in the internal computation of their contents and the internal reactions to their changes.

This section will present examples of how virtual trees can be used to build interactive applications, and how coherent reactions provide the key properties that make it work. The following examples continue from those of the previous section to build a simple web application for task management. We will omit many of the details of configuring a realistic web application. Assume that an HTTP server responds to GET requests on a certain URL by returning the value of the global variable currentPage, which is expected to contain a tree structure encoding an HTML page. We first define a form to display a task.

```
37  > currentPage = Grid[
38        ["name"; TextControl(task2.name)];
39        ["start"; NumberControl(task2.start)];
40        ["length"; NumberControl(task2.length)];
41        ["end"; NumberControl(task2.end)]]
42  =
```

name	task2
start	3
length	1
end	4

The Grid function on line 37 will generate the proper HTML for a table-based layout, the contents of which are specified inside square brackets and spread over the following four lines. Square brackets contain sequences of semicolon-separated values. (When a sequence follows a function call it becomes the first argument of the call, reducing bracket nesting.) The sequence elements are expected to be the rows of the grid, and are each specified with square-bracketed pairs of values. Each of these pairs contains a string label and a TextControl or NumberControl function which generates the HTML for a text input control. The resulting HTML is rendered on line 42.

Virtual trees supplant the template languages [14, 43] of most web application frameworks, avoiding their jumble of clashing syntaxes and semantics. Derivation expressions like Grid and NumberControl return arbitrary tree-structured values in-situ, allowing a seamless mix of literal structure and calculated content.

The above example hard-wires the reference to task2. To fix that we can define the TaskForm function below that takes a task as an argument.

```
43  > TaskForm: Fnc{
44      task: task1
45      val = Grid[
46        ["name"; TextControl(task.name)];
47        ["start"; NumberControl(task.start)];
48        ["length"; NumberControl(task.length)];
49        ["end"; NumberControl(task.end)]]}
50  > currentPage = TaskForm(task: task2)
```

Line 43 defines TaskForm as a variant of Fnc, the prototype of all functions. The next line defines task as an argument of the function, with a default value of task1. Functions define their value with the **val** field, as done on line 45. The function is called on line 50 to produce the same result as the previous example. The derivation of the form is diagrammed in Figure 4.

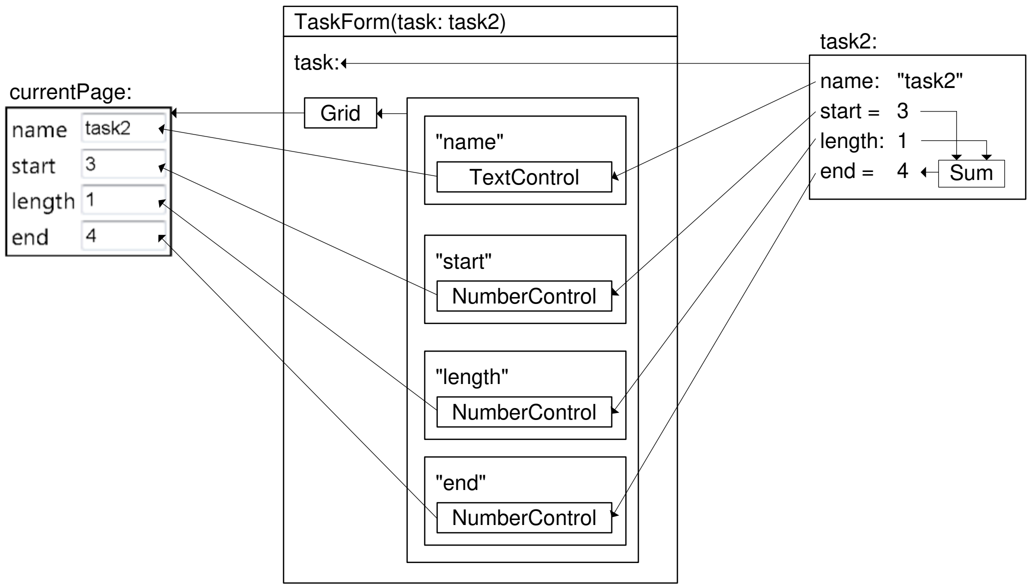

Figure 4. TaskForm derivation.

3.1 Input

To explain how user input is handled, we first examine how NumberControl is implemented:

```
51 > NumberControl: Fnc{
52     val: xmlNode{
53       tag: "input",
54       type: "text",
55       name: ^,
56       value = DecimalString(#1)}}
```

Line 51 shows that NumberControl is a function whose value is an xmlNode structure. Each xmlNode contains a tag field to specify the name of the XML tag it represents. XML attributes are stored as fields of the node. The name attribute is set on line 55 with ^ to be a reference to the xmlNode itself. This reference will be serialized into HTML as a string encoding the location of the node in the state-tree. Finally the value attribute of the input tag is derived with DecimalString to be a string representation of #1, the first positional (unnamed) argument to NumberControl.

The user can change some of the fields in the form and and submit it back to the server. The server will receive a POST request containing a set of name/value pairs for the input fields, and will deserialize each name into the location of the corresponding xmlNode in order to assign its value field. Unchanged values are discarded. The changes will propagate into the DecimalString derivations, which react by parsing the string back into a number, which then propagates into one of the linked fields of task2. These reactions simply reverse the direction of the derivation arrows in Figure 4. The combination of derivation and reaction play the role of the Observer pattern [23] and bidirectional data binding [41] in many application frameworks.

3.2 Causal error handling

Truth lies within a little and certain compass, but error is immense. – William Blake

What if there is an error, say the user enters "foo" into the end field? In that case, the DecimalString reaction would declare an error and abort the input transaction, discarding all changes. That would be too abrupt: we want to catch the error and reply to the user showing the submitted values with associated error messages. This is done by the following revised version of NumberControl.

```
57 > NumberControl: Fnc{
58     val: xmlNode{tag: "span"}[
59       control: xmlNode{
60         tag: "input",
61         type: "text",
62         name: ^,
63         value = Latch(DecimalString(#1), fence: userErrors)};
64       ^contol.value@error]}
```

Here the input field is wrapped inside an HTML span tag defined on line 58. The square brackets supply the contents of the span node, which contains the input field and a following error message. The input field is defined as before, except that on line 63 the call to DecimalString has been wrapped inside a call to Latch. Latches catch reaction errors. Normally the latch has no effect: it passes through the value of its argument. But if the value changes and the argument reacts with an error, the latch traps the error and prevents the transaction from being aborted. Latching permits errors to be reported to the user as follows.

The erroneous value that triggered the error is *latched* as the value of the Latch function. The error itself is recorded in the error field of the Latch. Assuming for simplicity that the error value is a textual message, the error is reported in

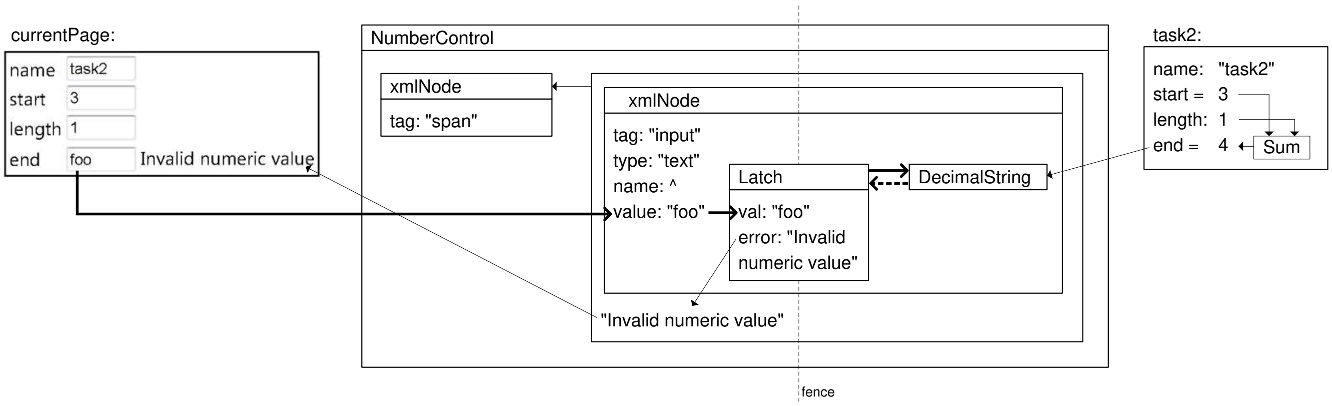

Figure 5. Detail of reaction error latching. Dashed arrow is error reflection.

the form on line 64 using the syntax control.value@error which refers to the error field of the Latch function. (The @ symbol is a path separator like the dot that is used to access a derivation function from the location it derives.) Figure 5 zooms in on the reaction error handling for the end field, showing how the latch catches and publishes the reflected error.

Latches are grouped into *fences*, specified with the latch's fence argument on line 63. The userErrors fence is used here to control the handling of all user-generated errors. Latches and fences change the way errors are handled. When a reaction declares an error, it is *reflected* back down the chain of reactions that triggered it. The error will be trapped by the first latch it encounters along the way. If the error reflects all the way back to one of the original input changes then it is fatal and the entire input transaction is aborted.

When an error is latched, all of the latches in the fence are activated. All changes caused by those latches are aborted, whether erroneous or not. Only changes inside the fence are committed. In this example, any other input field changes made at the same time as "foo" are preserved, but do not pass through into the task2 structure. A fence creates a boundary for staging changes and localizing errors. If crossing the fence led to an error, everything is rolled back to the fence but no further. Multiple errors at multiple latches can be trapped simultaneously by a fence. The next time a change hits one of the latches in the fence, all its latches will be released, and they will resubmit any changes they are holding. It is also possible to explicitly release a fence and discard all its latched changes.

Error handling is the source of much complexity in interactive applications. Conventional exception handling is problematic because the call stack at the time an error is detected does not always tell us how to handle it. Instead we have to carry around contextual information to attribute errors to user actions and set them aside for later processing. We have to ensure that errors are caught before state is left invalid, or else repair it. To handle more than one error at a time requires adding paths for continuing gracefully from an exception without triggering a cascade of redundant errors. Error handling in Coherence is better suited to the needs of interactive applications. Errors are handled automatically based on causal relationships, not control flow. Multiple errors are automatically traced back to their causes, and erroneous effects beyond fenced state are automatically backed out.

3.3 Coherent constraints

We can now handle the motivating example of model constraints from the introduction. Supposing that every task belongs to a project, we can add a constraint that all tasks must complete by the end of the project.

```
65 > task1: {
66      project: project1,
67      name: "task1",
68      start: 1,
69      length: 2,
70      end = Sum(start, length) | Leq(., project.end) "Too late" }
```

This definition adds a field referencing a project on line 66, and on line 70 a constraint on the end field. The constraint consists of a | followed by a predicate to test and an error to declare if it the predicates is false. The predicate compares the value of the constrained field (referred to with a dot), to the end date of the project. Constraints are checked whenever a field is derived or changed. This constraint on the value of end implicitly depends on the values of start and length, so we are faced with the familiar problem of executing the constraint only after all changes to those fields have been executed. Coherence does the ordering automatically. If the constraint fails, the error is reflected back down the reactions that triggered it. In this example the error will reflect from task2.end, through the DecimalString function, and be latched on the end field in the task form as described above. The following page results:

name	task2	
start	3	
length	1	
end	1000	Too late

3.4 Virtual sequences

Sequences are structures whose elements are ordered but not named[3], represented textually with square brackets. As with other forms of structure, sequences can be virtualized, which is useful for interacting with collections of data as in databases. Assume that all task instances are stored inside the tasks sequence. The following function derives a form to display all the tasks in a list.

```
71 > TasklistForm: Fnc{
72     tasklist: tasks,
73     let header: ["name", "start", "length", "end"],
74     let Row: Fnc{
75       val: [
76         NumberControl(#1.name);
77         NumberControl(#1.start);
78         NumberControl(#1.length);
79         NumberControl(#1.end)]},
80     val = Grid(Append([header], Map(tasklist, Row)))}
81 > TasklistForm()
82 =
```

name	start	length	end
task1	1	2	3
task2	3	1	4
task3	5	1	6

Line 80 defines the form as a grid whose first row is header, and whose subsequent rows are generated by a call to Map. The header is defined on line 73 with a **let** that declares it to be a local variable (i.e. private field). The Map function is like the standard map function of higher order functional programming languages. Its first argument is a source sequence, tasklist, which defaults on line 72 to tasks. Its second argument is a function (Row) to map over the sequence. Its value is the sequence of values of the function applied to the elements of the source sequence. Assuming there are three tasks, the result is equal to [Row(tasks[1]); Row(tasks[2]); Row(tasks[3])].

The novel feature of Map is that it is reactive: each element of the derived sequence is itself derived by an instance of the mapped function (Row), which will react to any changes within the element. In this case, that causes any user edits in the form to be directed to the corresponding task instances (and for errors to likewise reflect back to the correct form fields).

Derived sequences also serve as queries that can be updated. The following example displays a list of tasks in a specific project.

```
83 > Pred: Fnc{val=Equals(#1.project, myProject)}
84 > myTasks = Filter(tasks, Pred)
85 > TasklistForm(tasklist = myTasks)
```

[3] Coherence unifies named and ordered structures with *positions* [19].

Line 84 derives a sequence with the Filter function, which as in functional languages takes a sequence and a predicate to select elements with. The predicate is defined on line 83 to select only tasks in the project myProject. Instantiating the form with the derived sequence allows direct editing of those selected tasks without any extra programming: the form and the query compose modularly. What's more, we can insert and delete within such a query. Here is a variant of the form with simple insert/delete capabilities.

```
86 > TasklistForm: Fnc{
87     tasklist: tasks,
88     let header: ["name", "start", "length", "end"],
89     let Row: Fnc{
90       val: [
91         NumberControl(#1.name);
92         NumberControl(#1.start);
93         NumberControl(#1.length);
94         NumberControl(#1.end);
95         Button{label: "delete", do=>{Delete(#1)}}]},
96     let insert = Button{label: "insert", do=>{tasklist <<}},
97     val = Grid(Append([header], Map(tasklist, Row), [[insert]]))}
98 > TasklistForm(tasklist = myTasks)
99 =
```

name	start	length	end	
task1	1	2	3	delete
task2	3	1	4	delete
task3	5	1	6	delete
insert				

A delete button has been added to each row, containing an action that deletes the corresponding task. In line 96 an insert button is placed at the bottom of the table to create a new task. The action {tasklist <<} creates a new element in the tasklist sequence.

Both insertion and deletion work correctly even when the form is displaying the virtual sequence myTasks. Inserting a new task in myTasks will result in a new task in the source sequence tasks with its project field automatically set to myProject in order to satisfy the condition of the filtering predicate.

The Filter function reacts in two steps (using a two step progression over the source sequence). First Filter maps all insertions and deletions in the derived sequence back into the source sequence (tasks). Then it sets the value of the filter predicate on all the insertions to true. The filter predicate in this case is an Equals function, whose built-in reaction to true is to set its first argument equal to its second. That has exactly the desired effect of setting the project field to myProject. One can specify a custom reaction on the filter predicate to achieve any desired insertion behavior in a query.

Updatable views are not new [3, 7, 13, 22], but they have required that the filter predicate be written in a restricted language that allows a decidable analysis to infer the truth-preserving changes. Coherence takes a simpler approach. Filters are written as normal functions in the programming language, and are not analyzed. Instead, their values are set to true and it is left to their reactions to behave appropriately.

Often the built-in reactions will do the right thing, as in this example, but there is no guarantee provided. Programmers can supply their own reactions to implement arbitrary behavior as in database triggers [24].

3.5 Virtual variants

A key implementation technique of Coherence is the ability to efficiently make virtual variants of trees, up to and including the entire state of the system. Only the differences are physically recorded, with the variant being lazily constructed on demand. Tree variation is more subtle than it may at first appear: trees can contain subtrees that are variants of other subtrees, allowing variants of variants; and trees can contain variants of themselves, leading to (lazily) infinitely deep trees. Tree variation is by itself Turing-complete. [18]

Changes made within a variant do not propagate back to the source, but instead accumulate in the list of differences. Changes to the source are visible in the variant except when they are occluded by differences. The Reset action will erase all the differences in a variant, setting it equal again to its source. The Release action removes all the differences recorded in a variant and applies them to its source.

Variants can be used as database transactions. If we place an entire database within a tree, a transaction is just a variant of it in which the application makes all its changes until executing a Reset or Release action.

```
> database: {tasks, projects}
> tran: database{}
> ...
> Release(tran)
```

Coherent ordering guarantees that transactional changes appear atomic, and further allows multiple transactions to be atomically committed together. Transactions can be nested by making chains of variants.

3.6 And so on

Much more is required to build modern production-quality interactive applications. The goal of Coherence is to provide all the capabilities of modern application frameworks in a dramatically simpler form, as features of the language semantics itself, rather than as an immense pyramid of complex libraries. This section has surveyed some of these features as a way to validate the practical utility of coherent reactions and virtual trees. Virtual trees are a surprisingly versatile building block, and coherent reaction gives them nice properties of modular composition. They are worthy of further evaluation.

4. Related work

This paper addresses a fundamental issue of programming languages: how to manage side effects. This issue has been researched so intensively that it is impossible to cite all the related work. Instead, I will offer a declarative specification of the related work. This specification extends the motivating example from the introduction into a challenge problem and applies two metrics. The challenge problem is that we have a variable number of data model objects (like tasks) which contain constraints, and we have a variable number of view objects (like forms) that alter the model objects. The views take user input data to dynamically choose a model object (for example using a spelling-similarity algorithm) and then make multiple changes to that object. The constraint must not execute until after any changes that affect it.

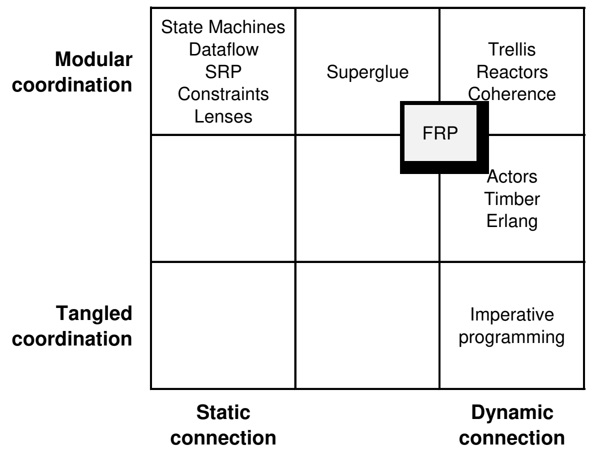

Figure 6. Related Work.

Figure 6 charts related work along two dimensions, which measure *modular coordination* and *dynamic connection*. Modular coordination is a measure of the coupling needed between the view and model objects to coordinate changes. Imperative programming fares poorly on this metric, for all the reasons discussed in the introduction. Dynamic connection is a measure of the ability to express the dynamic choice that connects views to models. Imperative programming scores at the top for dynamism: you can do anything you want, if you write enough code. Imperative programming thus goes in the lower right corner of the chart.

State machines [25] execute events instantaneously and simultaneously, in one stroke eliminating all coordination problems. But the price is that both states and events become statically structured. State machines do not allow variable-sized or variable-shaped structures, nor can one do arbitrary computations in them. You can not build a word processor purely with a state machine. After all, state machines can be compiled into gate arrays. Any language that compiles into hardware must sacrifice the essential softness of software. State machines go in the upper left corner of the chart, because while they coordinate change modularly, they can not implement the dynamics of the challenge problem. Practical languages based on state machines [38] do support more dynamic capabilities, but only by embedding them in an asynchronous imperative framework, which recreates the problems of coordinating events.

Dataflow languages [15, 34] are in the same situation as state machines. Synchronous Reactive Programming (SRP) [5, 11] was an inspiration for coherent reaction. SRP provides powerful features lacking in Coherence, including static analysis of cyclic dependencies and bounds on reaction time. But SRP is intended for embedded systems and intentionally limits itself to static state spaces that can be compiled into state machines or gate arrays. Like them it must go into the upper left corner of the chart.

Pure functional programming languages [28] address the problem of side effects by banishing them from the language. Monads [47] simulate imperative programming through higher-order constructions. The plethora of monad tutorials [39] witness that many programmers find them hard to understand. Yet they do not help solve the challenge problem, for the problems of coordinating side effects are recreated inside the monadic simulation of imperative programming.

Functional Reactive Programming (FRP) [12, 20, 29] entirely abandons the notion of mutable state. States become time-indexed sequences of values defined in terms of streams of input events. It is not clear exactly where FRP belongs on the chart, so it has been placed approximately. Some forms of FRP are capable of merging simultaneous events [33] to solve the challenge problem. But this solution comes at a heavy conceptual price. FRP inverts the normal order of cause-and-effect: effects are defined in terms of all causes that could lead to them. That approach seems to require that each model object know what views may connect to it in advance, breaking the modularity we seek. The response may be that we must learn different kinds of modularity, using some of the sophisticated abstractions proposed by FRP. At the least, FRP asks us to unlearn the common sense notion of mutable state. Coherence retains mutable state, abandoning only the Program Counter.

Superglue [36] combines FRP signals with OO abstractions to gracefully integrate with imperative code. Superglue connections are more modular and dynamic than conventional data binding techniques. Superglue can synchronize the events in the challenge problem. It lacks the computational power for fully dynamic connections.

Trellis [16] is a Python library that embodies the essential idea of coherent reaction, using transactional rollback to automatically order event dependencies. It appears to be the first invention of the idea. Coherence goes further by adding derivation and adopting the resultant model of computation as the semantics of a programming language. While Coherence was developed independently of Trellis, the prior work on Reactors [21] 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. Reactors support distributed asynchronous execution, which Coherence has not yet addressed. Both Trellis and Reactors can handle the challenge problem, placing them at the top-right corner of the chart.

The bidirectional execution of derivation and reaction resemble constraints. Sutherland first noted the usefulness of constraints in interactive systems [46]. They were combined with prototypes in the Garnet [40] system. Meertens [37] developed user-interface constraints based on bidirectional functions. Unlike Coherence, constraints can solve for fixpoints of cycles. But they are limited to static symmetric relationships, so they can not express arbitrary actions. The same is true for logical constraint languages like Prolog.

Lenses [7, 22] define a rich language of bidirectional functions over trees, supporting updatable views. Coherence builds upon that work. Lens are symmetric, and so can not express arbitrary reactions. Derivation and reaction combine into bidirectional functions, but are asymmetric, and use quite different semantics.

Actors [1] and related asynchronous languages [2, 6] improve the modularity of event handling in imperative programming, but can not implicitly coordinate the events in the challenge problem.

Coherent ordering shares some concepts with *serializability* [4, 45, 49] in transactional concurrency control, but they have different goals. The goal of concurrency control is to allow actions in different transactions to be interleaved so that they execute as if the transactions were executed serially rather than concurrently. Serializability is the source of the ACID properties of Atomicity, Consistency, and Isolation. But these properties do not apply between the actions within a transaction, which execute serially and have side effects upon each other. Coherence provides analogs of the ACI properties within a transaction, by ordering the actions so that they execute in *parallel*, as if they all occured simultaneously, with effects propagating instantaneously. Nevertheless, traditional concurrency control techniques might be adapted to finding coherent orderings.

	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 7. OO contrasted with Coherence.

5. 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 [35]

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 7 contrasts several other language aspects. The pattern that emerges strikingly matches the division of mental skills into *L-brain* and *R-brain* [31]. 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. Agha. Actors: A Model of Concurrent Computation in Distributed Systems, 1986.

[2] J. Armstrong. Erlang-A survey of the language and its industrial applications. In *In INAP'96-The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, 1996.

[3] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4), 1981.

[4] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

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

[6] A. Black, M. Carlsson, M. Jones, R. Kieburtz, and J. Nordl. Timber: A programming language for real-time embedded systems. Technical report, OGI Tech Report CSE 02-002, 2002.

[7] A. Bohannon, J. A. Vaughan, and B. C. Pierce. Relational lenses: A language for updateable views. In *Principles of Database Systems (PODS)*, 2006.

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

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

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

[11] 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.

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

[13] U. Dayal and P. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 8(3), 1982.

[14] P. Delisle, J. Luehe, and M. Roth. JavaServer Pages Specification v2.1. Technical report, Sun Microsystems, 2006.

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

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

[17] 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.

[18] J. Edwards. First Class Copy & Paste. Technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory TR-2006-037, May 2006. URL http://hdl.handle.net/1721.1/32980.

[19] 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.

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

[21] 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.

[22] 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.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[24] H. Garcia-Molina, J. Ullman, and J. Widom. Database systems: the complete book. 2008.

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

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

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

[28] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

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

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

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

[32] 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.

[33] W. Jeltsch. Improving Push-based FRP. In *Ninth Symposium on Trends in Functional Programming, TFP*, 2008.

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

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

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

[37] L. Meertens. Designing constraint maintainers for user interaction. *Manuscript*, 1998.

[38] S. Mellor and M. Balcer. *Executable UML: A foundation for Model-Driven Architectures*. Addison-Wesley, 2002.

[39] Monad tutorials timeline, April 18 2009. URL http://www.haskell.org/haskellwiki/Monad_tutorials_timeline.

[40] B. Myers, R. McDaniel, R. Miller, B. Vander Zanden, D. Giuse, D. Kosbie, and A. Mickish. The Prototype-Instance Object Systems in Amulet and Garnet. In *Prototype Based Programming*. Springer-Verlag, 2001.

[41] A. Nathan. *Windows Presentation Foundation Unleashed*. Sams, 2006.

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

[43] T. Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web*, 2004.

[44] 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.

[45] D. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems (TOCS)*, 1(1), 1983.

[46] I. Sutherland. Sketchpad, a man-machine communication system. In *Proc. AFIPS*, 1963.

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

[48] 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.

[49] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann, 2002.