Harmonious Authorship from Different Representations (Work in Progress)

Antranig Basman[1], Colin Clark[2], and Clayton Lewis[3]

[1] Raising the Floor - International
`antranig.basman@colorado.edu`
[2] OCAD University
`cclark@ocadu.ca`
[3] University of Colorado, Boulder
`clayton.lewis@colorado.edu`

Abstract. We describe the Infusion system, which is a library, language system or *integration domain* implemented in JavaScript, as well as a mentality and model for thinking about the expression of end-user applications. We promise that this system will bring together the worlds of different kinds of users engaged in different tasks at different times, and allow them shared authorial access to the same artefacts which are presented to each in a notation appropriate for them.

1 Introduction

Differing notations bring different affordances — and are suited for different audiences and different tasks(Blackwell & Green, 2003). For example, notations with low *viscosity* might be appropriate during initial development of a new system, whilst others with few *hidden dependencies* might be more appropriate during maintenance. Those with powerful *abstractions*, might be suited for experts, whilst others with good *visibility* might be better suited to novices or end-users. Traditionally, the choice of notation for a particular task implies more than a skin-deep commitment to a particular style of representation and way of working. For example, the choice of a conventional programming language such as Java or Haskell, based on the core representation of a stream of textual characters forming its source code, strongly limits the kinds of alternative notation which can be provided for other tasks and audiences. Correspondingly, the choice of a visual programming idiom such as Scratch(Maloney, Resnick, & al., 2010), Blockly(Google Developers, 2015), or Max/MSP(Cycling 74, 2007), cuts off the potential for engaging with audiences familiar with the power of traditional text editors and IDEs.

Our work for some years on the Fluid Project has been to build a system, Infusion, which aims to produce not just a single "middle way" between such extremes of notation, but also schemes for navigating between different notations in which "the same artefact" might be expressed. This will naturally involve some compromise between the needs of different audiences since, as in our examples above, the gap between the notational worlds of the visual and non-visual is not simply a matter of notation. The differences between the structure and referential style of, say, a Java program and a Scratch program are too profound to allow one to be usefully transformed and represented in the style of another.

Infusion evolves through repeated cycles of experimentation, validation and rationalisation, guided by some core heuristics. We still can't clearly see the forms of notation that can deliver on the aims we have just described — but we have made some crucial architectural decisions which put us at variance with existing popular notations, without which we believe that these aims cannot be achieved. These focus on the use of what we call *aligned, publically addressable state*, an idiom we will enlarge on in later sections.

A clear source of inspiration for Infusion is taken from the highly successful "evolved" solutions embodied in web technologies — we claim both the document object model (DOM) and representational state transfer (REST) idioms as embodiments of the aligned state idiom just described. As well as being inspired by the web, Infusion is built for the web — it is a library of standard JavaScript that can be included in any modern browser, and harmoniously coexists with applications written in standard markup and widgets. It is also suitable for standalone JavaScript runtimes such as `Node.js`.

2 Short guide and example

We'll work through a simple application encoded by an Infusion *component tree*. This will be considered first from the point of view of a user (author) named User A, employing traditional text-based development tools, and then from the point of view of other participants, Users B, C and D. We'll then consider the kinds of interactions they might share through the application, and the potential lifecycles of these interactions.

2.1   Brief guide to the structure of an Infusion application

An Infusion application consists of blocks of *configuration* expressed in the JSON serialization format, together with a collection of *globally namespaced*, ideally *pure function* expressed in the *base language*, which in our implementation is JavaScript. The configuration is organised as a set of globally named elements which are known as **grades**, which fulfil a few of the traditional roles of *types* in other systems, but fail to qualify in other areas. The configuration part of the system, since it consists of pure *state* aligned with a natural coordinate system, is ripe for transforming, expressing, and authoring in a variety of forms.

2.2   A small example involving relay

The *model relay system* is the configuration used to set up permanent, possibly transforming, relationships between different bodies of state. This kind of capability is also currently comprised under today's descriptions of *reactive systems*, particularly seen in the so-called *functional reactive programming* (FRP). In Figure 1, we'll set up a small system consisting of two pieces of state linked by a transforming relay, held in two different components, and then show how we can interact with it from the base language (JavaScript).

```
1   fluid.defaults("examples.simpleRelay", {
2       gradeNames: "fluid.component",
3       components: {
4           celsiusHolder: {
5               type: "fluid.modelComponent",
6               options: {
7                   model: {
8                       celsius: 22
9                   }
10              }
11          },
12          fahrenheitHolder: {
13              type: "fluid.modelComponent",
14              options: {
15                  modelRelay: {
16                      source: "{celsiusHolder}.model.celsius", // IoC reference to celsius model field in the other component
17                      target: "{that}.model.fahrenheit",      // could be shortened to just "fahrenheit"
18                      singleTransform: {
19                          type: "fluid.transforms.linearScale",
20                          factor: 9/5,
21                          offset: 32
22                      }
23                  }
24              }
25          }
26      }
27  });
```

Fig. 1. Short example showing a transforming relay from view of User A – temperature conversion

To start with, it's worth noting that so far our design consists of no base language code whatever. A single function call, `fluid.defaults`, is necessary to register the configuration with the system, but in other styles of interaction, for example the *Nexus* described in section 4.2 even this can be dispensed with. We'll need to execute some further base language code to create an instance of this system and experiment with it, but one can imagine that this also could be dispensed with in other visual/non-visual authoring environments which might feature, for example, a graphical "playground" in which instances can be set up and torn down by direct manipulation (see Figure 3).

In Figure 2 follows a sample conversation that we will have with the system. To start with, we will "decorate" the base system with some *model listeners* which will react to changes in the model values and report on them. We can do this i) without further application code, and ii) without needing to modify the above definitions. After that, we will use the language-level API to trigger modifications to the values and hence the reports.

This shows that the relay has set up a *lens* between the state held in the two components. The relay operates from the point of construction onwards — and ensures that the model constraint is satisfied by the initial system as well as with respect to modifications at either end of the relay. This relationship will persist until one or other of the related components is destroyed — which they might be, for example, with a call to `tree.celsiusHolder.destroy()`. As well as tearing down all relationships, this will also remove the instance from its parent, as required by the *cellular model* described in section 3.3.

2.3   The application from different points of view

The presentation of the application above is most suitable for a particular category of user, User A, who finds it convenient to use traditional software development tools based on text buffers and function calls. We'll now explore how we envisage this short application snippet could be shared with users of other kinds — for example, User B, who prefers a visual "boxes and wires" environment (mocked up in Figure 3) allowing development using direct manipulation by mouse, and the closely related User C, who would prefer a topologically identical environment, but instead mediated by speech and keyboard, in the style of the "nonvisual visual programming" environment presented in (Lewis, 2014). Finally, we imagine User D who is cast in the traditional role of an "end user" — they simply use the provided functionality of the app, with the simple interface shown in Figure 4.

```
1   // Designate a "decorated variety" of our simpleRelay type which will log messages on model changes
2   fluid.defaults("examples.reportingRelay", {
3       gradeNames: "examples.simpleRelay",
4       distributeOptions: [{ // options distributions route options to the subcomponents in the tree compactly
5           record: {
6               funcName: "fluid.log",
7               args: ["Celsius value has changed to", "{change}.value"]
8           },
9           target: "{that celsiusHolder}.options.modelListeners.celsius"
10      }, {
11          record: {
12              funcName: "fluid.log",
13              args: ["Fahrenheit value has changed to", "{change}.value"]
14          },
15          target: "{that fahrenheitHolder}.options.modelListeners.fahrenheit"
16      }]
17  });
18  fluid.setLogging(true); // send any logging output to the console
19  // Construct an instance of our decorated tree type
20  var tree = examples.reportingRelay();
21      // This will immediately report:
22      // Celsius value has changed to 22
23      // Fahrenheit value has changed to 71.6
24  tree.celsiusHolder.applier.change("celsius", 20);
25      // Celsius value has changed to 20
26      // Fahrenheit value has changed to 68
27  tree.fahrenheitHolder.applier.change("fahrenheit", 451);
28      // Fahrenheit value has changed to 451
29      // Celsius value has changed to 232.7777777777778
```

Fig. 2. Example of operating a transforming relay by User A - output is shown in comments
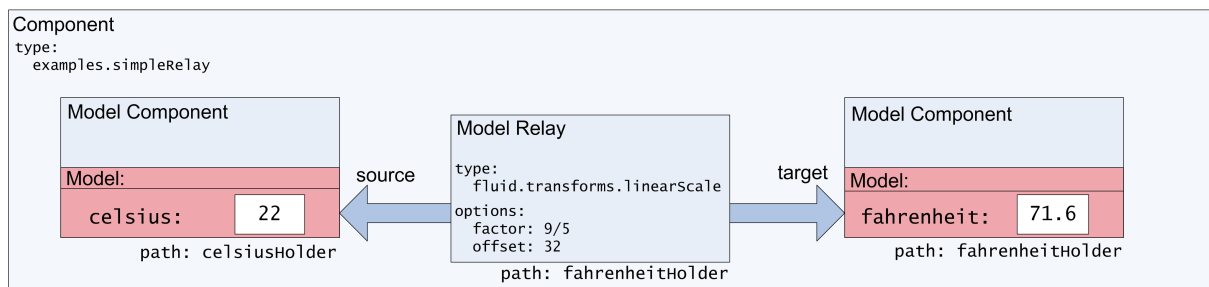

Fig. 3. Mockup of visual editing environment for authoring temperature conversion tree shown in Figure 1

Fig. 4. Simple HTML GUI for end user of temperature conversion tree shown in Figure 1

```
1   // Designate a "decorated variety" of our simpleRelay type which produces a live HTML interface
2   fluid.defaults("examples.relayApp", {
3       gradeNames: ["fluid.viewComponent", "examples.simpleRelay"],
4       components: {
5           celsiusField: {
6               type: "fluid.uiInput",
7               options: {
8                   model: {
9                       value: "{celsiusHolder}.celsius"
10                  }
11              }
12          },
13          fahrenheitField: {
14              type: "fluid.uiInput",
15              options: {
16                  model: {
17                      value: "{fahrenheitHolder}.fahrenheit"
18                  }
19              }
20          }
21      }
22  });
23  // Construct an instance of the application bound to the current HTML document's body element
24  var app = examples.relayApp("body");
```

Fig. 5. Decorating the model skeleton from Figure 1 to produce a simple HTML interface (markup not shown)

The notation/interface shown in Figure 3 contains the same information as that in Figure 1 (as would user C's hypothetical non-visual representation). Since this information has been expressed in the form of *aligned state*, we can directly correlate parts of these interfaces together, as well as user actions directed at them — we speak more about this kind of alignment in section 3.2. Because of this correlation, we plan for these interfaces to be usable simultaneously, to author one and the same underlying "application". Another result from this correlation is for user D's view, "the actual application itself" to be the target of authoring actions, in the style of Self's "the thing on the screen is the thing itself"(Ungar & Smith, 2013) model. This could be enabled by a "progressive disclosure" UI exposed, perhaps, to only some users in some contexts, allowing access to a progressively rich set of editing primitives drawn from the worlds of users C, B and A. The underlying application would be "live", to the extent that any of the participating authors introducing, say, a fresh temperature field showing values in

Kelvin, all views would update to show it (user D's only if a matching UI component were provided for it) — as well as, through the same underlying state-directed idiom, the current temperature value that a user had entered into any live embodiment of the application (for example, user B or D) would be preserved, and shown in the new temperature scale too.

2.4 The link to Inclusive Design

Our work on Infusion has been inspired the requirements of Inclusive Design, which requires applications to be freely adaptable to meet the requirements of users with different notational requirements, whether these are prompted by cognitive, physical, interactional or other differences. It's crucial that this can be done in an aligned way, such that the complete community of users sharing a particular requirement can share a particular coordinate, relating their embodiment of the notation to that used by another community. Our simple application can be seen as a direct example of such notational accommodation, in that a value (attached to a source of state in the world) is rendered to one user in one scale (Celsius), and another user in another (Fahrenheit). New adaptations can be freely introduced and removed from individual systems, without disturbing the wider community of cooperating users.

2.5 "One person's excess intention is another person's secondary notation"

A crucial requirement in order to meet our goal of harmonious authorship from different notations, is the construction of notations that are as free as possible from the expression of **excess intention**. Excess intention results when the notation we have available unavoidably captures more than what we intend to express in our design. Traditional programming languages, especially procedural ones, are famously rich in excess intention — some of which are being recognised and combatted by newer notations, others of which are not. Here are two examples we have characterised:

Sequential Intention — Imperative programming languages unnecessarily force the creator to commit to an exact sequence of executed instructions, which is usually far in excess of the real requirements underlying the goals they are interested in. This is a criticism that is broadly acknowledged, and some responses to it are becoming widespread — for example, as expressed in the model of *dataflow programming*, or in *monadic* styles of packaging control flow.

Artefact Boundary Intention — Object-oriented languages force the designer into a single, exhaustive decomposition of their domain of interest into a non-overlapping collection of *objects* with well-defined names, properties, relations and contracts. However, another view of exactly the same domain by a creator with different aims, skills or preoccupations might very well decompose it into an entirely different set of entities — which, with luck, at least bear a strict hierarchical relationship with those from the first view, but perhaps may not.

There are other intention excesses we could talk about, but each raises the same issue. In transforming from one notation to another, one must somehow capture all that is "excess" from one viewpoint with respect to another, and store it somewhere as an annotation to the structure — in exactly the same way one would be required to capture a *secondary notation* that had been attached in a notation, to preserve it during a stage of processing that was blind to it. This hugely complicates the design of tools manipulating such notations, because such notation is not in fact truly secondary. Even though the author was uninterested in it, this fact cannot be deduced from the notation, and the artefact cannot function without some expression of it. The lines of code in a function body must be supplied in *some* order — some of which are invalid, some of which are valid but don't capture the author's intention, and only a few of which are actually permissible equivalent expressions.

There is a link between this observation on secondary notation, and the concept already established in the literature of *redundant recoding*. (Fitter & Green, 1979) observe that the information encoded in a notation can be more easily grasped if it is encoded redundantly, and preferably in dissimilar modalities — for example colour or indentation as a perceptual mode, as opposed to textual structures or correspondances as a symbolic mode. Our focus on elimination of excess intention, in favour of expressions in an *aligned core* of state indirectly improves the scope for redundant recoding. Each of the styles of notation in section , given they can be interconverted, provides a source of redundant coding either within the same notation, or else by highlighting the corresponding elements in a different notation in an interface where more than one is presented at the same time.

3 Theoretical underpinning and links to existing paradigms

3.1 The role of programming languages and computational power

It is arguable whether the system we will present here is best described as a "programming language", a "framework" or some other thing. It shares clear characteristics of both. The best designation that we have found so far is that of an *integration domain* (Kell, 2009), which is an arena for the naming and scheduling of effects, computations and their units of organisation, rather than an system in which computation is expressed directly.

This issue, we feel, has long misdirected the field — since every notation which has been put into the role of "programming language" has been put under immediate pressure to demonstrate that it can express any computation ("is Turing-complete") in order to qualify for this role. On the contrary, an integration domain, as noted by Kell, can easily be endowed with lesser computational power, and we argue, should be so. It is crucial, for example, for the transparency and responsiveness of authoring tools, that relationships between parts of a program can be determined by the exercise of limited computational power. The Self family of languages emphasise the importance of such responsiveness for the feeling of authors that "the thing on the screen is the thing itself"(Ungar & Smith, 2013). A system or language whose structure implies the potential for unbounded computations (for example, those of a complex type system such as ML or Haskell) directly fights this aim. Such type systems, if provided, should be an optional addition to the system just for the use of a particular audience. Recent work on "gradual typing"(Siek & Taha, 2006) has tended in this direction, but so far there is little work on systems promising multiple independent, completely optional type systems for the same artefacts.

3.2 The first-class role of state, and transparent access to it

We promote the use of *transparent, publically addressable state*. The Infusion system should maintain all state — not just that on behalf of its users, but any that it maintains for its own book-keeping — in public view, with each piece of state available through an utterable[1] public address. This is at odds with both *object-oriented* and *functional* programming, which insist that the state which the application manages on behalf of users must be hidden from view, either through *data hiding* in the former paradigm, or *prohibition of side-effects* in the latter.

Publically addressable state is the touchstone of the prevalent REST(Fielding, 2000) style of conversation or API for web applications, and this analogy has guided our development since the start. REST stands for **representational state transfer** — describing a conversation where *state is represented*, rather than opaque tokens representing mere *behaviour* or *methods* as is common in procedural or object-oriented API contracts, and also where *state is transferred* — that is, that the representation is an *exhaustive* summary of the state that can be used to moved it from place to place.

The manifest nature of public state is crucial for many of the most successful embodiments of end-user programming. For example, in the spreadsheet paradigm, the programming surface consists purely of values in a grid. Each grid element has a well-known and mostly stable *public address* which can be used to access its value. Unfortunately from here on, the spreadsheet idiom starts to fall down, since any programming directives which are issued must skulk in a "hidden world" behind each cell, unaddressable either as a whole or in part. There have been efforts to address this deficiency within the spreadsheet paradigm in particular(Burnett & al., 2001). In general, we conclude that public addressability of all design elements is crucial in order for a design to allow good *visibility* and a lack of *hidden dependencies* when required.

3.3 A system inspired by the web, and built for the web — IoCSS selectors

The web represents the most highly successful "evolved strategy" for dealing with the problem of distributed and shared authorship. Whilst it appears to fall short of what are claimed as its antecedent blueprints, for example, in Ted Nelson's elaborate hypertext system Project Xanadu(Nelson, 1982), as well as being regularly claimed as a deficient abstraction by object-oriented and functional programmers alike, we feel that there is a great deal to study, admire, and learn from the solutions and strategies that it embodies.

Together with REST, discussed in section 3.2, another successful idiom that is essential for the day-to-day running of the web is CSS, the scheme familiar to designers and developers alike for describing the styling information applied to web pages. CSS fills a crucial role in brokering between distributed authors of "the same document" who live in different communities, with differing workflows and tools. The space of DOM elements in a web page is a shared authorial space that must be malleable in the face of demands of varying strength from different ends of the process (design and logic). The space of CSS selectors can be "negotiated" in that the requirement to identify a particular piece of the document could be met "opportunistically" by choosing a selector which matches it contingently and unstably, or by arranging/negotiating to alter the document structure to allow a selector to match it more stably and semantically.

Analogously, Infusion implements a selector system that can be used to flexibly refer to components within an application's component tree. We refer to this system as *IoCSS*, named after the framework's role as an "Inversion of Control" system. This implies that what has been previously thought of as "an application" has been endowed with a regular but free-form *cellular* structure. In the case of an Infusion application, the *cellular unit* is the *component*, rather than as it is with an HTML document the *DOM node*. The affordances of an Infusion component are unusual set against those of typical units of software designs, given that they may be

---

[1] All state in a system has some kind of address, but in practice not all such addresses are *utterable* by ordinary (that is, not specially privileged through forming part of the compiler, runtime or operating system) users or authors. For example, state held in a *function closure* is held at an address that cannot be named from programming language code. This implies that intentions held by users about such state cannot be encoded and acted upon

freely embedded recursively, and that further subcomponents may be injected into existing parents without their "knowledge" or disturbing the design. In object terms, Infusion components offer the possibility for *containment without dependency*, which is not possible in an object-oriented system.

Once we have the cellular structure in place, we now need some *landmarks*. In the DOM, these are provided by *CSS class names*, *tag names* and other well-known DOM attributes. In an Infusion application, these are provided by the *context names* which can be derived from the *grade names* attached to a component (which we first met in section 2.1) and the *member name* used to embed it in its parent.

IoCSS selectors are most often used with the `distributeOptions` construction which forms the framework's "polymorphism at a distance" scheme. Options material can be routed from any one source in the tree to any number of targets elsewhere in it. This scheme can be seen as similar both to *advice* offered in an *Aspect-Oriented Programming*, as well as encodings of a `diff` between values of state held within a revision control system.

3.4 Guide to Terminology

Term	Correlates	Distinction and Similarities	Intention and Advantages
Grade	Type/Class	Rather than establishing *contracts* or describing *storage*, a grade is a block of (JSON) configuration with a globally-qualified name which is merged in an aligned way with others to produce a description from which component instances can be built. Grade names can also be used as *landmarks* (*context names*) in order to bind segments of IoCSS selectors.	The use of grade-based descriptions reduces *excess intention* in descriptions of parts of implementations. The run-time structure of an instance is much more closely tied to the authoring-time structure, allowing for the "notation" of authors and users to be directly corresponded.
Model	Model (MVC Programming) / Model (Model-based development/MBD) / Behaviour (Functional Reactive Programming/FRP)	Infusion *models* encode mutable state in a JSON-equivalent form. Taken together with the associated model relay rules, these can constitute a model from the MBD point of view, since the space of model states can be deduced. Finally, the stream of values of a model over time can be compared to an FRP *behaviour*, transduced into other streams via transforming relay rules.	Similar to the use of grades, Infusion models minimise *divergence* between run-time and authoring structures. They also aid liveness and transportation of applications — it should be possible to effectively move an application between systems or users by transmitting just its models
Options Distribu-tions	Advice (AOP)/Diff (VCS)	An options distribution, like an aspect-oriented programming "advice", allows an existing application (component tree) to be modified by an author from the outside - that is, they can derive a variant application without modifying the expression of the original author. Unlike an advice, distributions have the same structure and syntax as ordinary configuration.	Since options distributions form a closed system, it is clear how multiple authors can collaborate on the same system, and multiple modifications competing to target the same piece of the design can have their relative priorities negotiated. This also implies that the same authoring tools can be used to write and check distributions as well as ordinary configuration

Table 1. Guide to terms used in this paper and relation to common forms

4 Current Work and Future Developments

4.1 The Preferences Editing Framework

As an example of a real-life view application built using Infusion, Figure 6 shows an instance of our *User Interface Options tool*, itself an instance of our *Preferences Editing Framework*. This application can be dropped into any web page to allow the user to customise the presentation of the page — for example, by selecting a custom font size, line spacing, contrast colour scheme or other accessibility adaptations.

This UI and its substructure are composed using Infusion's model-based approach from declarative materials similar to those in section 2. This implies that integrators can mix and match different panels into their UI, and customise all aspects of the existing ones, by editing JSON documents rather than writing code. We will build a self-editing view that will allow end-users and integrators to achieve this directly, in the style of the Morphic user interface tool for the Self language described in (Ungar & Smith, 2013).
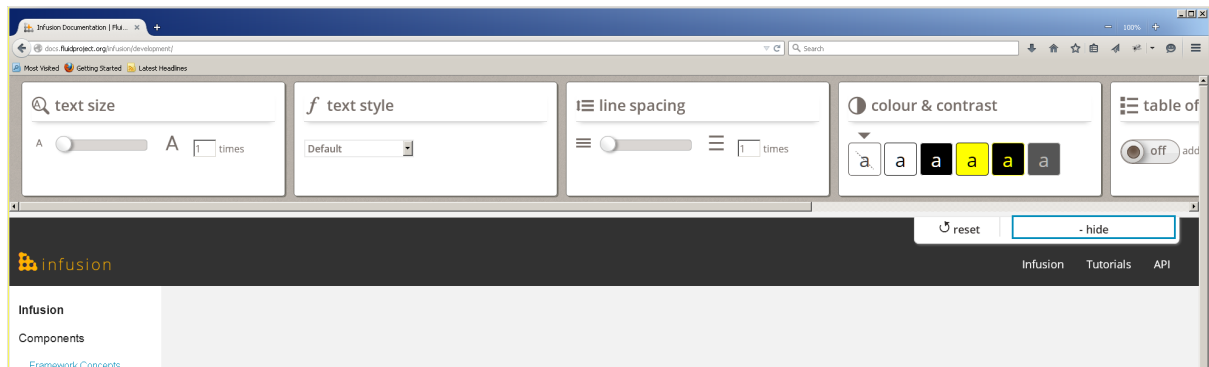

Fig. 6. Screenshot of a Preferences Editor instance built using Infusion, running on our documentation site

4.2 The Nexus - Support for Live and Distributed Authoring

For the EU project "Prosperity4All" (P4A - see `http://www.prosperity4all.eu/`), part of the overall Global Public Inclusive Infrastructure project (GPII - see `gpii.net`), we will be developing a portable and self-contained embodiment of the framework's facility as an integration domain named the *Nexus*. A Nexus consists of a standard instance of the Infusion system, supplied with HTTP and WebSockets endpoints which allow all of the core lifecycle actions to be operated remotely as well as via local function calls.

Since new "types", instances, and their relationships can be introduced into the system by pure web conversations with payloads of pure state, we have an ideal environment for sources and sinks of state to discover each other and to set up (possibly transforming) relationships between their respective bodies of state.

We plan to use this system to construct novel aggregations of raw devices to form *assistive technologies* (ATs), that will be able to adapt any means of input that a user finds appropriate onto any addressable target "application" or other sink of state such as an output device. The decomposition of updates from a text buffer into constituent Nexus messages will also be useful in other environments. In working with the Flocking(Clark & Tindale, 2014) system for audio synthesis on the web, we plan to produce a system which will close up the gap between the nature of *performance* and *score* by treating both as harmoniously cooperating elements on a common footing in a sea of state.

References

Blackwell, A. F., & Green, T. R. (2003). Notational systems - the cognitive dimensions of notations framework. In J. M. Carroll (Ed.), *HCI Models, Theories and Frameworks: Towards a Multidisciplinary Science.* Morgan Kaufmann.

Burnett, M., & al. (2001). Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *Journal of Functional Programming*, *11*(2), 155-206.

Clark, C., & Tindale, A. (2014). Flocking: A framework for declarative music-making on the web. In *ICMC 2014 - 40th International Computer Music Conference.* Athens, Greece.

Cycling 74. (2007). Max/MSP: History and Background. Retrieved from `http://web.archive.org/web/20090609205550/http://www.cycling74.com/twiki/bin/view/FAQs/MaxMSPHistory`

Fielding, R. (2000). *Architectural styles and the design of network-based software architectures* (PhD thesis). University of California, Irvine.

Fitter, M., & Green, T. (1979). When do diagrams make good computer languages? *International Journal of Man-Machine Studies*, *11*, 235-261.

Google Developers. (2015). Blockly: Language design philosophy. Retrieved from `https://developers.google.com/blockly/about/language`

Kell, S. (2009). The mythical matched modules: overcoming the tyranny of inflexible software construction. In *OOPSLA '09 proceedings* (p. 881-888). ACM.

Lewis, C. H. (2014). Nonvisual visual programming. In *Psychology of Programming Interest Group Annual Conference* (p. 129-134).

Maloney, J., Resnick, M., & al. (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education*, *10*(4).

Nelson, T. H. (1982). *Literary machines*. Mindful Press.

Siek, J. G., & Taha, W. (2006). Gradual typing for functional languages. In *Scheme and functional programming* (p. 81-92). ACM.

Ungar, D., & al. (2014). The Self Handbook. Retrieved from `http://handbook.selflanguage.org/4.5/`

Ungar, D., & Smith, R. B. (2013). The thing on the screen is supposed to be the thing itself. Retrieved from `http://davidungar.net/Live2013/Live_2013.html`