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.

Keywords: POP-II-A. individual differences; POP-II.C. cognitive dimensions, data flow, visual languages; POP-I.A. programming economy; POP-I.C. web

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 referred to. 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 some users (authors) named Users A, A′ and 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 A small example involving relay

The *model relay system* is 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 JavaScript, which we'll call the *base language*. The components and relay are expressed as configuration in JSON, referencing function and component definitions in JavaScript via strings representing their *globally namespaced* names.

```
fluid.defaults("examples.simpleRelay", {
    gradeNames: "fluid.component",
    components: {
        celsiusHolder: {
            type: "fluid.modelComponent",
            options: {
                model: {
                    celsius: 22
                }
            }
        },
        fahrenheitHolder: {
            type: "fluid.modelComponent",
            options: {
                modelRelay: {
                    source: "{celsiusHolder}.model.celsius", // IoC reference to celsius model field in the other component
                    target: "{that}.model.fahrenheit",       // this reference could be shortened to just "fahrenheit"
                    singleTransform: {
                        type: "fluid.transforms.linearScale",
                        factor: 9/5,
                        offset: 32
                    }
                }
            }
        }
    }
});
```

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

To start with, it's worth noting that our design so far involves no JavaScript code. 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 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 5).

We now imagine that another user, User D, *decorates* this definition with some further elements (some shown in Figure 2) that can turn it into a live HTML interface, allowing a further user, User E, to use the interface (shown in Figure 3). User E, cast in the traditional role of an "end user", can type numeric values into either field and see the opposite field update with the corresponding value in the other scale.

```
// User D designates a "decorated variety" of our simpleRelay type which produces a live HTML interface
fluid.defaults("examples.relayApp", {
    gradeNames: ["fluid.viewComponent", "examples.simpleRelay"],
    components: {
        celsiusField: {
            type: "fluid.uiInput",
            options: {
                model: {
                    value: "{celsiusHolder}.celsius"
                }
            },
        fahrenheitField: {
            type: "fluid.uiInput",
            options: {
                model: {
                    value: "{fahrenheitHolder}.fahrenheit"
                }
            }
        }
    }
});
// Construct an instance of the application bound to the current HTML document's body element
var app = examples.relayApp("body");
```

Fig. 2. Decorating the model skeleton from Figure 1 to bind to a simple HTML interface (markup not shown)

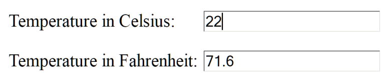
Temperature in Celsius: 22
Temperature in Fahrenheit: 71.6

Fig. 3. Simple HTML GUI for end user (User E) of temperature conversion tree shown in Figure 1

2.2 Decorating the skeleton for console interaction

In this section, we will imagine two further users decorating the same application skeleton in Figure 1 to perform interactions from the browser console. User A′ will decorate the base 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, user A″ will use the language-level API to trigger modifications to the values and hence the reports. These interactions are shown in Figure 4.

```
// User A' designates a "decorated variety" of our simpleRelay type which will log messages on model changes
fluid.defaults("examples.reportingRelay", {
    gradeNames: "examples.simpleRelay",
    distributeOptions: [{ // options distributions route options to the subcomponents in the tree compactly
        record: {
            funcName: "fluid.log",
            args: ["Celsius value has changed to", "{change}.value"]
        },
        target: "{that celsiusHolder}.options.modelListeners.celsius"
    }, {
        record: {
            funcName: "fluid.log",
            args: ["Fahrenheit value has changed to", "{change}.value"]
        },
        target: "{that fahrenheitHolder}.options.modelListeners.fahrenheit"
    }]
});
fluid.setLogging(true); // send any logging output to the console
// User A'' uses the grade of User A' to construct an instance of our decorated tree type
var tree = examples.reportingRelay();
  // This will immediately report:
  // Celsius value has changed to 22
  // Fahrenheit value has changed to 71.6
tree.celsiusHolder.applier.change("celsius", 20);
  // Celsius value has changed to 20
  // Fahrenheit value has changed to 68
tree.fahrenheitHolder.applier.change("fahrenheit", 451);
  // Fahrenheit value has changed to 451
  // Celsius value has changed to 232.7777777777778
```

Fig. 4. Example of operating a transforming relay by Users A′ and A″ — output is shown in comments

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 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 original authoring of the application was by 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 how the authoring of 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 5) 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). These would lead to the same experience by the end user E in Figure 3.

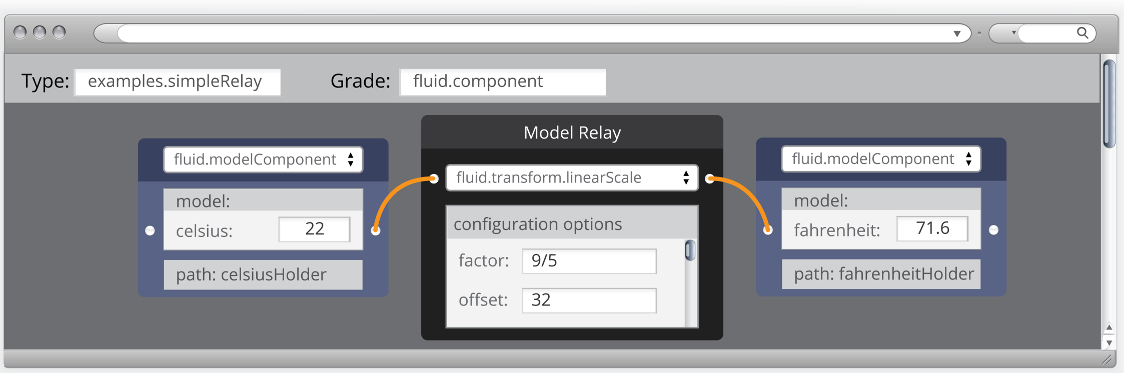
Type: examples.simpleRelay
Grade: fluid.component

Fig. 5. Mockup of User B's visual environment for authoring temperature conversion tree shown in Fig. 1

The notation/interface shown in Figure 5 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 E'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, if any of the participating authors introduce, say, a fresh temperature field showing values in Kelvin, all views would update to show it (user E'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 A'' or E) would be preserved, and shown in the new temperature scale too.

2.4 The link to Inclusive Design

Infusion is based on Inclusive Design practices, where software should 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 need 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. More substantial adaptations can be introduced, such as accommodating special devices or modes of input and output.

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 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 may or may not bear a strict hierarchical relationship with those from the first view.

Other sources of intention excess raise similar issues. 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.

Our system addresses intention excess issues in a few ways. Our choice of configuration primitives is guided by an autoethnographic process that attempts to explain our intentions when writing base language code. Configuration styles which fail to do this economically are discarded, and the process continues to iterate. In general, intention cast in the form of aligned, transformable state naturally involves less excess than that in terms of other primitives (such as function or object definitions) — since, for example, both sequential and artefact boundary intention is minimised. Finally, the coordinate system that the state is endowed with provides a natural set of "hooks" on which to hang secondary notations as the primary notation is transformed between representations.

3 Theoretical underpinning and links to existing paradigms

Our 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.

3.1 The role of programming languages and computational power

It is arguable whether Infusion 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 also 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 it 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 traded, which represent 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 move 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. (Burnett & al., 2001) address this deficiency within the spreadsheet paradigm in particular. The public addressability of all design elements is crucial for a notation 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 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 and the *member name* used to embed it in its parent. Some roles for IoCSS selectors using these landmarks are summarised in Table 1.

[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 outside the closure. This implies that intentions held by users about such state cannot be encoded and acted upon.

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 Distributions	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

Examples of real-life applications built using Infusion can be seen at `http://fluidproject.org/` — in particular our *User Interface Options tool*, itself an instance of our *Preferences Editing Framework* embedded on our documentation site at `http://docs.fluidproject.org/infusion`. This tool can be dropped into any web page to allow the user to customise the page's presentation — for example, by selecting a custom font size, line spacing, contrast colour scheme or other accessibility adaptations. 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*. 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 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.*

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.

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., & 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`