Harmonious Authorship from Different Representations

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

[1] Fluid Project, OCAD University, Toronto, Canada
amb26@ponder.org.uk / cclark@ocad.ca
[2] 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 on different tasks at different times, and allow them shared authorial access to the same artefacts which are presented to each in a notation appropriate to them and their situation. We explain the importance of strictly limiting the computational power consumed by such expressions for the impression of liveness and transparency during the authorial process, as well as the corresponding importance of making directly visible, manifest, and addressable, the *state* which an application is managing on behalf of its users. The device known as a *lens*, imported from the discipline of *bidirectional programming*, is a crucial metaphor and structuring device which sets up a permanent, possibly transforming, relationship between the artefacts at its endpoints. Processes and users can do work at both ends of the lens using a representation which is appropriate to them. We show some examples of this approach and describe the direction of current and future work.

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, some 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. Some, 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, Rusk, Silverman, & Eastmond, 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 at the Fluid Project has been to construct a system, Infusion, which offers 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 — 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. Every system constructed so far strongly betrays its heritage of having been initially conceived as a "visual" or "textual" system from the start, and no amount of tooling or authoring support can ever bridge such a divide.

In development since 2007, Infusion still counts as a "work in progress" — although several aspects of its nature have become over the years, and our group uses it every day for their real-world programming tasks, there is a lot of ground still to cover in making its idiom consistent in all areas, and tackling the tough architectural challenges related to managing "programs" which can be authored live by a distributed group of authors using dissimilar representations. In addition, work on visual and remote tooling for the system is only just beginning. Infusion is a system inspired by the web and designed for the web — it consists of a standard JavaScript library which can be included in any modern standards–compliant browser, and harmoniously coexist with applications written in standard markup and widgets. It is also suitable for standalone JavaScript runtimes such as `node.js`.

1.1 Quick guide to the structure of an Infusion application

An Infusion application consists of blocks of *configuration* expressed in the JSON(“Introducing JSON”, n.d.) serialization format, together with a collection of *globally namespaced*, ideally *pure function* expressed in the *base language*, which in our language is JavaScript. The configuration is organised as a set of globally named elements (in the same global namespace as the function) which are known as **grades**, which fulfil a few of the traditional roles of *types* in other systems, but fail in several respects to qualify as types as well as taking on some non-traditional functions which we will discuss later. The configuration part of the system, since it consists of pure *state* which comes with a natural coordinate system, is ripe for transforming, expressing, and authoring in a variety of forms.

1.2 The role of the “base language”

The part of the design expressed as code in the base language is as usual only suitable for being authored by software experts using traditional software tools — but this part is expected to be a small part of an end-user design, and is expected in the long run to consist of a very finite collection of well-known and well-curated instances. As time passes, the need to involve a software expert in any particular design is expected to become rarer. In parallel, as time passes, we expect the features used within the base language gradually become more and more impoverished — for example, we at the earliest stage abandoned the features of JavaScript used for “object” formation using prototypal inheritance — following that, the features used for forming closures for expressing higher-order functions — and are in the process of abandoning those used to achieve side-effects through assignment and then finally those used to express looping and general control structures other than conditionals[1]. The resulting “impoverished base language” will then itself become far more amenable to novices, authoring using non-traditional tools and will in essence lose much claim to identity as being JavaScript at all — the same core subset used for producing a limited range of expressions designating pure functions with no control structures beyond conditionals and structural recursion might as well be taken from any language. We expect that any dialect used for expression actual *computations* will be ejected from the system and be treated with the same caution that is currently applied in pure functional languages for code with side-effects or performing I/O in general. We’ll talk more about the *avoidance of computation* in section 2.1.

1.3 “One man’s excess intention is another man’s secondary notation”

A crucial requirement in order to meet our goal of harmonious authorship from different notations, is the construction of notations 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 he is 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* (which we comment in section 2.2 is one possible cast for some of Infusion’s function), 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 — if lucky, which at least bear a strict hierarchical relationship with those from the first view, but perhaps not so. One answer to this kind of of issue is named within the Design Patterns community as the *Facade Pattern*(Gamma, Helm, Johnson, & Vlissides, 1994) — however, in all traditional notations, the construction of a Facade involves fresh programming language code, which implies expert intervention, a high maintenance burden and all that goes with it. And a Facade cannot accommodate the situation where one of the new boundaries pass through the middle of a former artefact — one cannot construct a Facade which encloses “half an object”.

There are other 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

[1] “...cursed their feet, cursed the ground, and vowed that none should walk on it again” — Douglas Adams, *The Restaurant at the End of the Universe*, Chapter 10

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 as expressions, some of which are valid but don't capture the author's intention, and only a few of which are actually permissible equivalent expressions.

An important characterisation of our mission, then, in constructing Infusion, is to construct a notation which is as deficient in *excess intention* as possible. As examples of this:

— it must not commit the author to particular sequences of execution of actions above those which are necessary to express their intention

— it must not commit the author to particular boundaries of artefacts in their domain, but permit "Facades" to be constructed by mere reference, rather than fresh implementation, as well as allowing the boundaries of these to penetrate existing artefacts.

1.4 The historical and methodological process of constructing Infusion

This bipartite model consisting of "configuration over a base language" is also crucial to the methodology by which Infusion was developed and will continue to be developed. At its inception in 2008, Infusion applications consisted largely of standard JavaScript code, loosely organised around records containing "options", of which the JSON configuration simply represented *default values.* This historical structure gave rise to the name by which the directives holding configuration were and still are issued to the system, `fluid.defaults`. As year succeeded to year, the power and capability of the configuration system steadily increased, and the number, diversity and complexity of the functions that required application code steadily decreased. At any time, the space of possible application function consisted of an "explained portion" which could be expressed as configuration, and the unknown "unexplained portion" which still required conventional code. Development of the framework proceeded by a scientific processes whereby a generalisation was proposed that might reduce part of the "unexplained portion" to practice — and was validated if we found that future designs that had not been anticipated could productively make use of the feature or configuration style. Features which were found to be used only in one or few designs, the ones that prompted the invention of the feature, were pruned from the framework.

This "scientific" methodology, where features require to be validated by real-word practice, is at odds to the style of development typically used in new languages or frameworks, which typically proceeds in a "clean-room" style driven only by imagination, insight and memory. The designer takes a handful of features which appears to them to be desirable on some theoretical or historical grounds, and tries to spin a language or environment around them with as much consistency as possible. The trajectory of such systems is typically also hostile to the "scientific" process — either the language gains no currency, in which case it is simply discarded, or else it becomes widespread, in which case its evolutionary potential becomes strongly limited. Radical shifts in idiom are precluded because of the large and in many cases unknowable swathes of user application code which would be broken by the change.

Our process has been favourable in contrast, owing to the fact that application designs have been retained and curated within the community. The task of rewriting old designs to new standards has been undertaken internally — implies that the costs could also be tracked, as well as retaining the ability to evaluate whether an old design has indeed become more economical when expressed in new notation. This process of *autoethnography* has been necessarily slow, but we believe necessary to the success of any notation in the long term. Many other notations, we feel, could have been enormously improved had they gone through a much more lengthy incubation and maturing process before being publically adopted — and conversely, if designs that had been quickly abandoned had instead been tended, improved and polished.

2 Theoretical underpinning and links to existing paradigms

The development of Infusion has been influenced by that of numerous other systems. It partakes of a little of each of the flavours of these systems, whilst retaining its own coherent and consistent idiom. It's easiest to describe the new in terms of the familiar — in this section we'll describe some of these ancestors and neighbours and what we have taken from each.

2.1 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 in some clear characteristics of both. The best designation that we have found so far is that of an *integration domain* (Kell, 2009) — an arena for the naming and scheduling of effects, computations and their units of organisation - rather than an area in which computation is expressed directly. This issue, we feel, has long been one giving rise to confusion and misleading results in 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 computation 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) - and 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 against this aim. Such type systems, if provided, should be an optional "bolt-on" 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.

2.2 Other roles for Infusion

The original billing for Infusion was as an "Inversion of Control" (IoC) or "Dependency Injection" (DI) (Fowler, 2004) system. It certainly still directly fills that role — during instantiation, the system resolves symbolic references to state and other entities around the component tree in a data–driven way. Unlike other such systems (such as Spring, Google Guice, etc.) which resolve such names in a flat context, in Infusion, every component, as well as acting as a container for other components, can act as a context which influences the resolution of names in nearby components. This data–driven process gives rise to another view of Infusion, that of a "dataflow programming language". As well as this static (that is, each name resolves just once only) system which operates at construction time, there is a more familiar counterpart which operates dynamic dataflow for mutable state during the main lifetime of the components. This mutable state is held in areas of each component known as *models*. Since the transition rules guiding the propagation of state via dataflow are available to be read from the component's configuration and to be used for inference by authoring and validation tools, this also positions Infusion within the area of *model-driven development*(Schmidt, 2006). Finally, when these model areas are combined with configuration binding them into the view layer of the browser's DOM, the system can be viewed as a *Model-View-Controller* (MVC) framework, although the controller layer in an Infusion application has no code associated with it.

The `model` area of an Infusion component adopts the popular JSON serialization format as its model for state - this state consists of Numbers, Booleans and Strings, stored in Arrays (indexed by Numbers) and Objects (indexed by Strings) — these latter two containers can be organised into hierarchical structures forming composite models. As we will describe in the following section, as well as ensuring that user-managed state takes this form, we are also working to make the entire records of the system runtime itself available in a similar form, although making these records serializable in such a straightforward way will be elusive.

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

We present here what has emerged as the key idiom which distinguishes our system from others in numerous areas. Both of the current mainstream programming idioms, *object-oriented programming* and *functional programming* 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. We promote entirely the opposite - that the system maintain all state, not just that of its users, but any state which it maintains for the book-keeping purposes of its own runtime, in public view, with each piece of state available through an utterable public address.

There is not room in this paper to explore all the consequences and implication of this decision, and it is one that has proved the hardest to put across the meaning and nature of when describing our goals. For now, we present three links between this decision and other bodies of theory.

Firstly, 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**, and each of these words carries crucial and essential meaning. Firstly that state is represented in the conversation, rather than opaque tokens representing mere *behaviour* or *methods* as is common in procedural or object-oriented API contracts, and secondly that it is *transferred* - that is, that the representation of the state is to some extent an *exhaustive* summary of the contents of one of the interacting systems, that can be moved from place to place.

Secondly, the crucial importance of such *exhaustive representation* for the *perceived liveness* of any proposed system. In terms of our discussion of the Self language(Ungar & al., 2014), for example, one fault possessed by Self as well as systems influenced by it, such as JavaScript, is the divergence between what could be called *constructional type* and *live type*. This is exemplified in JavaScript, for example, by the non-identity between the object property named `constructor`, holding the name of the creator function used to instantiate an object, and the one named `__proto__` which identifies its prototype chain. In Self, the actual runtime contents of an object are influenced both by its *prototype* and its *traits* which lead to definitions for a collection of its *slots* which may be changed at runtime. The result of this is that the in-memory representation of an object incorporates many hidden pieces of bookkeeping information (*hidden dependencies*) which guarantee that the object can never be made straightforwardly equivalent with a particular serialisation of it. Self provides the *Transporter* which exports objects to an essentially opaque format only readable by another Transporter (although by a well-documented algorithm), whilst JavaScript provides natively only for serialisation to JSON — which guarantees only to represent plain state rather than the live state of any objects using its inbuilt prototypal scheme.

Thirdly, 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 many efforts to address this deficiency, most notably in the PPIG community, but we will say for now that public addressibility of all design elements is completely crucial in order for a design to be able to exhibit good *visibility* and a lack of *hidden dependencies* when required — one can always choose to hide what might be visible in contexts where it is undesirable, but one cannot chose to make visible that which does not even have an utterable address.

It's worth observing that this model of public addressibility is directly analogous to the actual model operated by real computing devices. At the hardware level, all state is stored in an *address space* where very location has a stable, publically usable address (sometimes more than one). Each successive layer of software development has merely served to obscure the values of such a system, and create new locations (such as the interiors of objects, or of function closures) where state can be stored with unutterable names. This has been driven by the perceived virtues of *data hiding* or *implementation hiding*, where implementation units have been isolated from each other in order to promote the possibility of design consistency and heading off chaotic effects caused by unmanaged reads and writes. We believe that by rethinking application and language structure, and prohibiting the power to make unnanounced reads and writes to mutable state by application code, we can not only retain these virtues, but also recover the virtues lost at the lower levels, where the entire capabilities of a machine or a design were put at the disposal of the end-user.

2.4 A system inspired by the Web, and built for the Web — the IoCSS selector system

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.

We referred in the previous section to the importance of the REST idiom, first used to describe the stateless and resource-oriented nature of web protocols, to the design of our system. Another successful idiom which 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 for visual rendering. This scheme fills 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" which requires to 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 arranging/negotiating to alter the document structure if required in order to choose a selector which matches it more stably and semantically.

To our knowledge, this is the first time that the affordances of CSS selectors have been made available to the implementors of applications rather stylers of documents. By analogy, we have named this system of selectors ***IoCSS***, named after the framework's IoC role described in section 2.2. Naturally 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 document, 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 1.1) and the *member name* used to embed it in its parent. In addition there are two special context names, the *root context* holding the global root of the entire component tree represented by the forward slash `/`, and the *current context* represented by the name `that`. These appear since, unlike CSS names which universally address the target document from a vantage point outside it, many IoCSS selectors are attached to particular locations within the component tree and need to be interpreted relative to it.

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 system can be seen as analogous to a few other schemes and notations. Firstly, in the increasingly popular *Aspect-Oriented Programming* scheme, modifications to a program's structure known as *advice* can be targetted at it using a special language for these aspects. In the language of that community, the specification of the locations where these modifications are to be made is named a *pointcut*. Infusion options distributions have the advantage that no distinct language is required in order to express this "metaprogramming" — they consist of standard options material attached to ordinary components, and so can themselves be targetted by other distributions etc. without risking an troublesome infinite regress of level of metadescription and meta-advice. Secondly, if the bodies of state themselves are considered as documents, perhaps stored in a version management system, one might consider that the payloads attached to options distributions represent encodings of the `diff`s between values of state held at different versions. In this case, the "versioning"-like requirement arises from the differing requirements of two different audiences or users of the same artefact. We will see a simple example of such a compact encoding of differences in Figure 2 in the next section.

3 Some Small Examples

In this section, we'll give some small examples of expressions in Infusion, starting with a minimal "application".

3.1 A small example involving relay

The *model relay system*, which we alluded to in section 2.2 under the heading of "dataflow languages", is the configuration used to set up permanent, possibly transforming, relationships between different bodes 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).

```
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",       // could be shortened to just "fahrenheit"
                    singleTransform: {
                        type: "fluid.transforms.linearScale",
                        factor: 9/5,
                        offset: 32
                    }
                }
            }
        }
    }
});
```

Fig. 1. Short example showing a transforming relay

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. Naturally we will need to execute some further base language code to create instances of this system and experiment with them, 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.

The two fields `celsius` and `fahrenheit` as well as the `modelRelay` rule could easily have all been defined in the same, base component, leading to a one-component system rather than one with three — however, we have broken them out in this case to show the scheme by which we issue a plain IoC reference on line 16 of this example referring to one component's model from the other, and show IoCSS selectors in the following example targetting reporters at the components from the tree base.

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 (the **ChangeApplier API** — see `http://docs.fluidproject.org/infusion/development/ChangeApplierAPI.html`) to trigger modifications to the values and hence the reports.

```
// Designate 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
// 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. 2. Example of operating a transforming relay

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 2.4.

3.2 The Preferences Editing Framework

As an example of a real-life view application built using Infusion, Figure 3 shows an instance of our *UIOptions 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.

Rather than being constructed from raw JavaScript code, this UI and its substructure are composed using Infusion's model-based approach. The raw materials for such a UI consist of a *Primary Schema*, which is a standard JSON schema (see `http://json-schema.org`) describing the data types of the preferences being edited, and an *Auxiliary Schema* which is essentially a dehydrated form of Infusion component tree. 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. Our plan is eventually to build a direct-manipulation style 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 **??**.

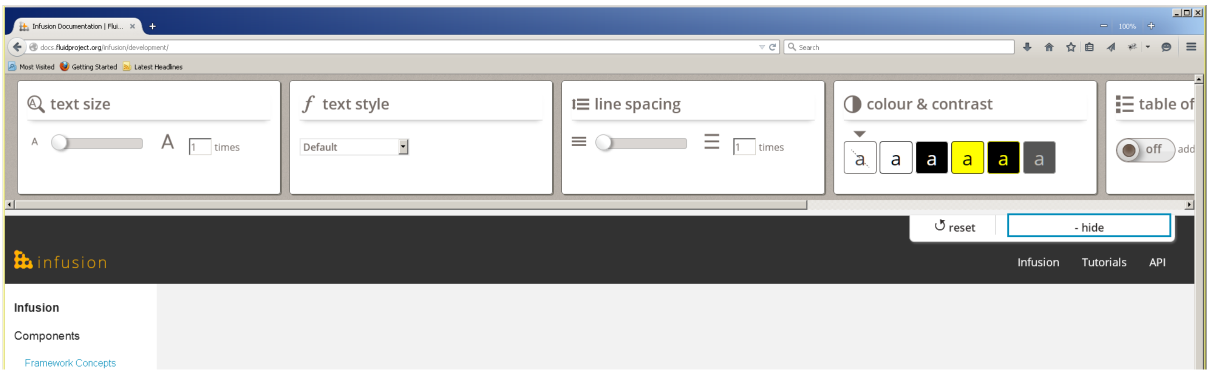

Fig. 3. Screenshot of a Preferences Editor instance built using Infusion, running on our documentation site

4 Current Work and Future Developments

4.1 Framework Improvements to Handle Asynchrony

A long-required facility in the framework is for all of the instantiation machinery to operate asynchronously, as well as to operate a lifecycle controllable across an entire component tree rather than points representing the lifecycles of individual objects or components. This will allow us the control we need to harmoniously operate the (at least) two dataflow-driven processes that we described in section 2.2 — the static process guiding instantiation, and the dynamic process guiding resolution of model state and constraints. In the current implementation these tend to tread on each other's toes since their properly distinct workflows end up triggering parts of each other irregularly during startup. Proper handling of asynchrony will also allow us to properly integrate external effects such as network traffic gracefully into the construction process, as well as bringing our system into alignment with a formalism that has long been attractive to us, the "Galois" system for the parallelisation of irregular algorithms(Kulkarni & al., 2008).

4.2 The Nexus

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 just consists of a standard instance of the Infusion system, but with instrumentation added to the instantiation machinery in order to provide standard HTTP and WebSockets endpoints which allow all of the core lifecycle actions to be operated remotely as well as via local function calls. For example, fresh defaults can be issued into the system by an HTTP PUT to `defaults/<component.name>`, or a fresh instance constructed at a particular path by an HTTP POST to `construct` with a JSON body holding path name, type name and options. Also, one can enter into a persistent conversation with any model in the tree by opening up a WebSockets connection to `models/<component.path>/<model.path>` — this starts a bidirectional flow in which outgoing model changes are reported in a live stream, as well as the system responding to incoming change notifications on the other leg of the connection.

Since both new "types", instances, and their relationships can be introduced into the system by pure web conversations with payloads of pure state, this provides an ideal environment for sources and sinks of state to discover each other, and set up (possibly transforming) relationships between their respective bodies of state. To the extent that this can be done without application code, this can be done freely by all kinds of citizens, although it requires for its setup that the sources and sinks themselves have been provided with relevant adaptors of their behaviour to the *integral state model* as well as the system to have all the relevant transformers already built into it.

We plan to use this system to construct novel aggregations of raw devices to form *accessibility technologies* (ATs), which will be able to adapt the means of input that a user finds appropriate onto any addressable target "application" or other sink of state such as an output device.

4.3 Support for Live and Distributed Authoring

The implementation of the Nexus will require ambitious improvements to the framework's instantiation model. We will need to support a much more permissive process for issuing updates to both the grade information as well as the component tree, in that any work unit submitted to the system will need to be backed out cleanly in the case it generates a failure, as well as being merged together with the effects of any concurrent updates in the case it is successful. This implies that not only the dynamic dataflow (between model state) but also the static instantiation dataflow will need to be made fully *transactional*, together with the scheme for issuing failure diagnostics back to the author. A standard runtime model of issuing a text diagnostic and terminating, or leaving the system in a potentially undefined condition after an assertion failure, is not acceptable. However, once successful, the Nexus system for updates will allow for many new kinds of authoring workflow, not just with respect to multiple simultaneous creators, but for the same creator with respect to their own past. Given we expect that consistent values for merged application state to be mechanically recoverable from a version history, the possibility is raised for "jamming live with the past" — that is, for recovering past application states from version history as part of a graceful and reliable process, rather than the current experience with version management tools which don't even guarantee to recover a syntactially valid program given two historical versions, and enmesh the user in painful and error-prone conflict resolution schemes. In working with the Flocking(Clark, 2015) 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.

Clark, C. B. D. (2015). Flocking: Creative audio synthesis for the web. Retrieved from `http://flockingjs.org`

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.

Fowler, M. (2004). Inversion of control containers and the dependency injection pattern. Retrieved from `http://martinfowler.com/articles/injection.html`

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley.

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

Introducing JSON. (n.d.). Retrieved from `http://www.json.org/`

Kell, S. (2009). The mythical matched modules: overcoming the tyranny of inflexible software construction. In *OOPSLA '09 proceedings of the 24th ACM SIGPLAN conference companion on object oriented programming systems languages and applications* (p. 881-888). New York: ACM.

Kulkarni, M., & al. (2008). Scheduling strategies for optimistic parallel execution of irregular programs. In *SPAA '08 proceedings of the Twentieth annual symposium on parallelism in algorithms and architectures.*

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education*, *10*(4).

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

Schmidt, D. C. (2006). Model-driven engineering. *IEEE Computer*, *39*.

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`