Software and How it Lives On - Embedding Live Programs in the World Around Them

Antranig Basman Raising the Floor amb26@ponder.org.uk
Luke Church University of Cambridge luke@church.name
Clemens Klokmose Aarhus University clemens@cavi.au.dk
Colin Clark OCAD University cclark@ocadu.ca

Abstract

Virtues beyond those that traditionally motivate live programming are needed to support lively and unbounded communities of authors collaborating by creating and using shared artefacts. We will argue for the importance of each element of an artefact's design to be *externalizable*, and introduce terms describing the function of parts of a fully capable live externalizable system (the *res potentia* and *res extensa*). We critique the standard presentation of live programming, situating it within a wider set of authorial values. We introduce the quantity of *divergence* of a programming language or system and explain the desirability of minimising it. We survey some existing systems through this taxonomy and speculate how future systems could improve on them.

1. Introduction

Liveness in software, as described by (Tanimoto, 2013), (Ungar & Smith, 2013), and others, is essential for bringing the affordances of reshaping software to everyone who uses software. Liveness by itself, however, is not sufficient to produce durable, sharable software artefacts supporting lively and unbounded communities of authors. We consider that the values of liveness are better understood when considered against the experience of networks of authors, rather than single ones. From this point of view, the strict values of liveness itself (that is, level of immediacy of perceived updates to the system) are less important than the authorial affordances of liveness, in terms of being able to use the system to author itself. In section 4.4.4 we critique (Tanimoto, 2013)'s taxonomy of levels of liveness, and point out that levels of liveness that he considers "higher" (the predictive levels L5 and L6) can be achieved without necessarily achieving the letter of liveness he considers prerequisite (immediate editing liveness at level L4). In sections 1.1 and 3, we situate what we consider underlies the real value of liveness in terms of the authorial cycle experienced both by single authors and networks of authors.

In this paper, we will introduce some new virtues which need consideration, as well as some new qualities, categories of design elements, and ways of looking at designs that they induce. These virtues support *externalization* — the ability for elements of a design to have mobility within a space of authors, to be shared and operated on by a variety of tools which provide "elite affordances" (version management, durability, unit and integration testing[1]) to those not traditionally considered part of the software engineering elite. We will say that a design, system or language fails to be properly externalizable if it, at runtime, exhibits *divergence* — a discrepancy between its bookkeeping, runtime state and the state with which it can be externally authored. We develop a taxonomy of system state based on (Kauffman, 2011)'s presentation of the terminology in (Whitehead, 1929), consisting of the *res extensa* and the *res potentia*. This taxonomy accounts for the authorial purpose and fate of system state in order to promote designs with smaller divergence.

1.1. Liveness and the Authorial Cycle

Here we outline the conceptual background for this paper, with a brief account of liveness and Whitehead/Kauffman's taxonomy.

(Ungar & Smith, 2013)'s axiom of live programming is that "the thing on the screen is the thing itself." There should be as little discrepancy as possible between the artefact shown to users and the system

[1] These are examples of affordances that are all taken for granted as available in any professional software development context, but are traditionally absent in user programming systems.

which can be used to build it. This implies, firstly, that a system can be authored in place — that is, authoring a system should not require intrusively tearing it down and rebuilding it, resulting in loss of access to the system during the interval it is being worked on. Secondly, it implies that all of the tools required to build and modify the system are delivered along with the system, and can be accessed through an interface appearing next to or on it.

Liveness has been enthusiastically accepted as essential by a small section of the programming community, but has not yet brought about a revolution whereby every piece of software that can be used, can also be modified, customised and shared. These are values that we take for granted with physical artefacts and tools, but are not yet widespread for their informational equivalents. In this paper we try to identify an authorial barrier which impedes the adoption of live systems, and consider ways that it might be removed.

In order to identify the nature of this barrier, we borrow some terminology from (Kauffman, 2011; Whitehead, 1929), originally used to describe the universe itself — this is natural since we expect that the function of software is to hold a mirror up to nature[2]. In Whitehead's model, reality consists of two Realms, the Possible and the Actual, in which Actuals give rise to Possibles, which in turn give rise to Actuals. We adopt this workflow as a paradigm for live programming, in which Possibles (considered as the design space of the system) give rise to Actuals (considered as running systems) through the process of execution. Then, the process of authorship, directed at the running system, as well as the process of Observation in general (interaction of the system with its context, the outside world) feeds back into the realm of Possibles.

In section 2 we'll introduce the key quality of *externalizability* in a design, which enables a particular, intimate and bidirectional relationship between Actuals and Possibles, and in section 3 split the realms of Actuals and Possibles into particular domains named from (Kauffman, 2011)'s treatment, which names Actuals as constituting the *res extensa* (extended matter, the realm of that which is), and Possibles as the *res potentia* (potential matter, the realm of that which might be), which we will split into potentia I & potentia II.

2. Externalizability, Documents and Models

For elements of a designed software artefact to be easily shared and authored by a variety of different tools, live or otherwise, it is essential that they are *externalizable* — that is, for these elements appear with the affordance of *documents*. Documents, in our presentation, are transparently addressable by a natural coordinate system, and represent a format in which an exhaustive summary of state in one part of a running system can be transported from place to place. A natural coordinate system is one that allows units of the implementation to be addressed in their position within the document's structure in such a way that combinations of expressions from different authors can be aligned and overlaid in a semantically meaningful way.

Documents with a basis for this idiom are described in (Fielding & Taylor, 2000)'s presentation of the REST idiom as an interpretation of the meaning and function of the web as operated by the HTTP protocol. In REST, an exhaustive representation of a resource is transferred during a conversation, rather than the more prevalent "API" or "message passing" style of conversation where transmissions consist only of answers to limited questions with an essentially arbitrary semantic. Fielding considers that documents should have a publicly intelligible textual semantic, but does not venture further into considerations of their support, via internal coordinates, for authorial networks.

We'll use the term *model* (in the sense of the MVC community, rather than the sense of the model-checking community) to refer to the part of a design imaged when a document is transferred. All elements of the *res potentia*, and some elements of the *res extensa* (those that we will simply name models) consist of "model material" and are hence directly serializable as documents in this sense.

[2]Hamlet: Act 3, Scene 2 — "playing, whose end ... is, to hold ... the mirror up to nature"

The traditional, meagre externalisation of a program in the form of its source code is unsuitable to participate in the lifecycle we outline in section 1.1, since source code is a very fragile notation for authoring by means other than a human being sitting in front of a text editor. It has only a very weak correspondence in structure with that of the running program, and cannot be reliably processed by tools other than its one intended audience — the programming language's compiler and its closely bound tools. One tradition, *Literate Programming* (Knuth, 1992), somewhat extends the audience for part of this text to include authors at design time. Specially formatted elements, embedded alongside elements of the source text, are marked for processing by a special tool chain which may either publish them as a static document describing the system, or else appear in a live interface assisting an author who is in the context of selecting between the elements designated by the source text. However, this text only has a unidirectional role in the authorial cycle — it consists just of human-readable text and can't assist the system to operate on itself.

3. Actuals and Possibles in Software Systems

The web as seen in its "Web 1.0" incarnation, as it originally emerged in the 1990s, traffics purely in documents in their own right — those which represent a rendered web page, which are serialized as HTML and in model form are represented as the DOM. We call such a "document in its own right", following Kauffman's presentation of (Whitehead, 1929), a *res extensa* (representing **what is**). It consists of a fully actualised system — in this case, the web page itself, displayed in a browser.

However, an executing application, rather than simply a markup document, requires more sophistication than this — whether we are considering general user applications, or the "Web 2.0" AJAX-enabled web applications that emerged in the 2000s. In a dynamic system, we need to describe the space of entities that may come into existence, in addition to those that are currently running. In traditional software engineering, this is achieved through the ability to define classes, types, or other kinds of implementation unit which can then be instantiated. We call this kind of material *potentia I* (representing **what can be**), forming the first category of the *res potentia*.

The second constituent of the *res potentia*, which we call *potentia II*, represents **what is to be**. This is the registry of user expressions — that is, the collection of instructions they have given in order to designate that elements of *potentia I* should be instantiated, and the addresses they should be instantiated at.

In existing programming systems, extensa, potentia I and potentia II are entangled together, and few to none of them are externalizable. Although some systems separate the extensa from potentia I, none to the knowledge of the authors separate potentia II from the others. The process of defining and instantiating objects occurs at the same level of system design, and the processes of designating an object to be instantiated and initiating the instantiation process itself are not separated. Some examples of these situations appear in section 7.1.

4. Divergence

The Web itself, considered as simple system presenting only documents using an extensa (separate from its dynamic functionality added on by scripts), can easily meet our criteria for externalizability. However, the requirement to present arbitrary applications, rather than simply content, creates significant problems. Any realistic system, in the pursuit of animating its *extensa*, must make use of significant amounts of internal book-keeping information — which in practice it will be hard to imbue with the semantics of documents. This information is held in hard-to-access implementation elements such as the compiler's runtime, the virtual machine or other libraries. This causes the state held in the extensa to *diverge* from a usably externalizable representation.

This problem is very likely insurmountable in its totality. However, certain choices of language and system design will tend to exacerbate or reduce the system's divergence. Systems with lower divergence will tend to be easier to author and work with through a variety of tools (for example, textually-directed tools, version management tools, or live tools presented at the system's own interface or remotely), since these tools can be targeted at the externalized, document form of the design, and hence address a greater

proportion of the real, executing design.

4.1. Divergence from the Stack

The divergence in a system or language typically also manifests itself as an obstruction to liveness itself — since the runtime or external tool has as a task to update the divergent material in some kind of consistent way with respect to the authorial updates. For example, (Church, Söderberg, Bracha, & Tanimoto, 2016) cite that a hazard in imbuing their highly dynamic system with liveness is the task of updating (or if impossible, destroying) any stack frames which are in the process of executing through an updated class (potentia I element). This highlights that the program stack is one of the most prominent divergent elements of most current language designs, and one whose impact has to be strongly restricted. An ideally non-divergent system would eliminate the program stack entirely as a runtime phenomenon — a tall challenge for a system which requires to remain intelligible and efficient.

4.2. Divergence from Object Representation

Dynamic and live programming languages typically expose facilities where the interfaces of objects can be customised at an instance level. Whilst Smalltalk (Black et al, 2011) frowns on this somewhat (methods are defined on classes, not on instances — however, method lookup can be dynamically intercepted), Self (Ungar, Chambers, Chang, & Hoelzle, 1991) provides primitives for dynamically adding and deleting slots from objects. After this has been done, the in-memory representation of the object has diverged from the one resulting from its hierarchy. Were the object destroyed and reconstructed, the authorial information would be lost. Both Self and Smalltalk provide a scheme for serializing and restoring objects and prototypes into persistence, known as the "object transporter". The files written by the object transporter do not qualify as documents in our taxonomy, since they have no natural coordinate system and cannot easily be manipulated by tools other than the virtual machine which produced them. The Pharo (Pharo, 2016) and Squeak Smalltalk (Squeak Project, 2016) projects cooperate by providing access to a DVCS system named Monticello (Bryant & Putney, 2016), which system is capable of interpreting the output of the object transporter on those particular Smalltalk dialects. The fact that this is a platform-specific facility rather than a generalised one highlights the fact that the virtual machine contents have not been externalized in the sense we have described, and that general text- or document-based version control systems such as git are not easily applicable. A Smalltalk or Self program is essentially a hermetic machine image — mixing the functions of an extensa (in that it is running) and a potentia (in that it is authorable).

4.3. Divergence from Event Listeners

Essentially any programmatic (as opposed to declarative) authorial action targeted at the *extensa* may result in divergence, but we'll round off this list of examples with a final prominent case — that of registration of event listeners. Traditionally, events represent a multicast publish/subscribe pattern which has some kind of incarnation in essentially every programming system, especially those which present some form of user interface. Event listeners represent a primary source of divergence because they are traditionally very hard to coordinatise. Programmatic systems tend to treat listener equality or identity based on the equality of function (or slot) handles which naturally leads to problems once part of a system enters serialisation outside a running VM. This issue can often lead to design puzzles in itself — for example, if a particular method or slot is added multiple times as a listener to an event, should it be notified multiple times? If it is then removed, should just one instance of the listener be removed, or should it be all matching ones?

4.4. Strategies for Minimising Divergence

No unified strategy for opposing divergence in all its forms can be mounted — since these forms are highly diverse, and do not stem from a single cause; instead, they each result from particular engineering tradeoffs, physical and cognitive ergonomics. Considering our sources of divergence above, we could consider, for example, the following kinds of strategies for reducing their design impact:

4.4.1. Fighting Divergence from the Stack

To reduce the impact of the stack as a source of divergence (that is, its tendency to interfere with our ability to author a running system as if it consisted of pure, externalized state), we could consider strategies like these:

Firstly, we could prohibit trying to author the system whilst the stack is not quiescent. This might for some applications, represent a significant loss of liveness (pushing us down from L4 to L3 or lower in (Tanimoto, 2013)'s taxonomy). However, the strict definition of L4 liveness has problems with respect to a fully collaborative system, which may be in progress with edits from multiple users concurrently. If we make the design choice (an expensive one) to fully isolate these transactionally, we may be able to finesse L4 liveness by claiming that the system "does not visibly have an execution in progress" if it is observed from outside the context of the relevant user's transaction.

Secondly, we may adopt execution strategies which tend to make for short stacks, or, more radically, abolish the use of call stack entirely. There is some precedent for this in ancient and modern asynchronous or message-passing architectures. For example, highly distributed systems such as Erlang partition work out to many thousands of "processes" (execution sites which may or may not correspond to distinct physical execution sites). As a result, Erlang programs tend to have much shorter, "broader" stacks. There is a clear relationship with our first stack elimination strategy if we imagine that a "process" takes the form of a forked avatar of a substantial part of the state in the system (during a transaction), which is capable of being joined back to it (if the transaction commits successfully). Other connections are with the "spatialised computation" of (Kulkarni et al., 2008)'s "Galois" framework, and with the rising tendency of JavaScript (and other) developers to implement architectures using *Promises*. Promises by convention (though not by necessity) resolve asynchronously, cutting the stack at the point they evaluate.

The costs of these strategies are to dilute the important virtues which the machine stack was designed long ago to embody. Firstly, the stack as an "explanation" for the system's current actions is a truly vital affordance when authoring — for example when debugging, or in the case of an error. Naturally tools and libraries could compensate for this loss, but they would have to exist, be ubiquitous and cheap, and of high quality — and it would be hard for them to be as ubiquitous and cheap as something which is forcibly embodied in the machine's own execution model. Secondly, the stack is a vital target for important classes of optimisations. A function call could never be inlined if it never occurred in the first place.

4.4.2. Fighting Divergence from Object Representation

It is possible that Smalltalk's discouragement of instance-based object modification stems from a taste against divergence. It certainly points the way towards a family of strategies for dealing with this problem — since the problem, after all, stems from the possibility that an entity with a "less utterable name" (an instance) diverges from one with a "more utterable name" (a class/prototype). If the design of the system forces the user to give a more utterable name to a unit as part of the authorial action of modifying it (or its class), divergence is headed off — since the crucial affordance required by document semantics is that the document is transparently coordinatised (section 2). In our taxonomy of state from section 3, forcing the authorial action to be ascribed to some named category of entity helps the system push responsibility for the edit from extensa up to potentia I.

Another strategy which assists here is to segregate the state held at a unit (object/component) into material which explicitly is considered "model state" (which is mutable, but has edits ascribed to potentia II), and the remainder, which is considered immutable, to which edits must be ascribed to potentia I. This increases the costs of understanding the system's design, given it represents a lack of orthogonality in its affordances, but resolves the problem in authorial meaning. This is the strategy currently adopted by the Infusion system.

4.4.3. Fighting Divergence from Event Listeners

Event listener divergence can only effectively be combated by removing the affordance to add or remove event listeners as a direct authorial action. As with other cases of divergence, we need to produce idioms for pushing the association of events with their targets up into potentia I. This implies that lifetime and reference for listeners are both managed by the system, in response to some form of declarative dialect encoding the relative positions and identities of the target and source. We can generalise this category of problem to the problem of maintaining any kind of binary relationship between pairs of objects in the system. The natural choice is to bring the relationship into existence at the time point where the later of the two objects has been constructed, and tear it down when the first of them is destroyed.

4.4.4. Queen of Sheba Adaptation

In Chapter 16 of his *Fusus al-Hikam* (The Bezels of Wisdom, written c. 1229) (Al-Arabi, c. 1229), the Sufi philosopher Ibn al-Arabi describes an encounter between Solomon and the Queen of Sheba (Bilqis) in which he causes her throne to appear before her, apparently displaced by thousands of miles in an instant. She comments on seeing it, "It is as if it were the same". al-Arabi analyses this incident from the point of view of doctrine of the "renewal of creation by similarity". Under this model, the contents of the universe may be being destroyed and renewed countless times each second through the will of Allah. This process is not evident to us because we, the observers, are destroyed and renewed along with it.

This suggests a simple test as to whether a system is divergence-free, as well as a cheaper means for systems to undergo adaptation in the face of authorial instructions than to implement full L4 liveness in the sense of Tanimoto. If we can destroy a component of the system and then regenerate it from its potentia I and potentia II records, in either an identical condition, or a changed one in order to reflect potentia II adaptations, the system must be divergence-free. If, in addition, we can arrange the system such that no observer could perceive the system in the intervening time (which capability is mostly implied by our treatment of transactions in section 4.4.1), we could say that the system (inefficiently) is both fully live and divergence-free. This scheme might involve less implementation effort than a more ambitious system that could compute the minimal differences between the two configurations of the *extensa* before and after the change, even if the latter would execute much more efficiently. This treatment suggests that the Tanimoto taxonomy has an anomaly at the point where the L4 liveness level is defined — in that we could easily imagine systems delivering L5 and L6 predictive capabilities even where the strict letter of L4 had not been observed. One such system is the Entelechy system described in (Church et al., 2016).

4.4.5. Tools or System

We will visit these kinds of considerations further when we look at particular examples of systems in later sections. However, we will note that they place a huge burden on the development of powerful and highly usable tools in order to compensate for the affordances which the divergence sources were originally designed to meet. In practice, appeals to the existence of such tools are far more frequent than the tools actually coming into being. However, it's clear that pushing forward the standard model of live programming, where "the tools are part of the structure of the system itself", has to be the model for resolving these problems — whilst, at the same time, avoiding the standard trap of live programming, whereby the running system itself becomes a hermetic, self-sufficient ecosystem with little reference to things which lie outside it. These "tools", when we build them, must be every bit as externalizable and divergence-free as the "system" that they arguably operate on.

4.4.6. Homeostasis rather than Execution

In order to enable a harmonious authorial cycle, we need to create programming systems in which the extensa, potentia I & II are cleanly separated. In particular, separating potentia I and potentia II creates a very different idiom for the function of the running system — rather than being given the task of *executing* the user's expressions (properly supplied as potentia II elements), the system instead is engaged in a process of *homeostasis* — that is, constantly choosing actions at its disposal in order to minimise what discrepancy there is between the extensa and the potentia from time to time. This model

of homeostasis also allows for *predictive homeostasis* — speculatively anticipating authorial expressions in order to achieve L5 and L6 liveness in Tanimoto's terms.

4.4.7. Mocks and a Taxonomy of Effects

A crucial requirement for the speculative execution of an L5/L6 live system is a clear schema identifying which implementation units have side-effects and which do not. Effects issued against the external world cannot be called back — effects issued against our own implementation might possibly be. Ideally, as well as being able to read metadata identifying an effect-laden component, the running system would also be able to locate a variant component that had some value of substitutable behaviour that was free of the side-effects. In the testing community, these implementations are known as "mocks". In traditional software development, mocks and test cases are considered part of a separate arena of expression and activity. Here we close the loop on another of the "elite affordances" that we identify in our introduction, unit and integration testing. A fully competent authorial system would make no sharp distinction between test cases and real execution, between mock implementations and real ones — since a speculative mock implementation executing at the behest of the future should be equivalent to a real implementation executing today.

The test cases for each artefact would be part and parcel of its packaging in the system, together with metadata allowing the system to identify one or more mock implementations that could be substituted for it in cases where its side-effects need to be limited as a result of speculative execution.

5. Webstrates

Webstrates (Klokmose, Eagan, Baader, Mackay, & Beaudouin-Lafon, 2015) is a Web framework designed to explore a software vision based on Alan Kay's early conceptualizations of interaction with a computer being interaction with a dynamic medium (Kay & Goldberg, 1977). Webstrates persists all changes to the Document Object Model (DOM) of any page, called a webstrate, served from the Webstrates server, and synchronizes these changes between clients of the same page. This includes changes made to inline JavaScript and Cascading Style Sheets (CSS). In effect Webstrates turns the DOM into a collaborative and malleable medium. Development happens inside the browser either through the developer tools of the browser, or through authoring webstrates that transclude other webstrates through the use of `iframes`. Klokmose et al (Klokmose et al., 2015) demonstrate how transclusion of webstrates can be used to create dynamic application-to-document relationships between webstrates, and allow users to collaborate on the same documents with personalized and radically different user interfaces.

5.1. Externalizability

Development with Webstrates follows the dogma that all important application state is be stored in the DOM. If this dogma is followed, application state will persist over a page reload and will synchronise between clients with the same webstrate open. The DOM can be serialised to HTML, this means that an application running on Webstrates can be serialised, deserialised and resumed in the same state (in principle, see section 5.3). It also means that the application state can be modified in its "dead" serialized from, and be the subject of some elite affordances like version management.

5.2. Actuals and Possibles

Webstrates uses the conventional JavaScript language and runtime, and the relationship between the actuals and possibles is therefore mostly conventional. Yet, part of the res extensa is expressed in the DOM, and is therefore inspectable, modifiable and serializable, and in Webstrates, collaboratively editable. The dynamic behavior of a webstrate is inlined as JavaScript stored in `script` nodes. Inlined JavaScript can be (collaboratively) edited at run-time, but must be re-evaluated by the client (either through reloading the page or using `eval`).

5.3. Divergence

Webstrates on the one hand exhibits a low degree of divergence, as most application state is represented in the DOM that can be losslessly serialized to human-readable and human-editable HTML. However, on the other hand the JavaScript runtime (objects, prototypes, functions, event handlers etc.) is not

present in the DOM, but is instantiated from JavaScript that typically is stored in script nodes. Run-time changes to e.g. a prototype object will not be serialized back into the JavaScript code it was instantiated from in the first place. This means that the run-time state of two clients of the same webstrate can differ, which can be exploited to create collaborative applications with relaxed WYSIWIS[3].

6. Dynamo

Dynamo (www.dynamobim.org) is a programming ecosystem for computational design, it is mostly used within computational architecture. It is a hybrid visual/textual and dataflow/imperative language using a conventional node and arc notation with dataflow semantics; it can also be seamlessly converted into DesignScript, a textual language for exploratory programming. Dynamo's primary uses are for exploring geometric forms and for automating Building Information Management (BIM) workflows, such as computing furniture needs for a building. As such it has a very strong connection to Autodesk Revit. Dynamo is an L3/L4 live system, with the level of liveness controllable by a checkbox in the interface. The default behaviour is that any changes to the program will execute as soon as they can. The system will re-execute automatically in response to slider changes — and more ambitiously — to changes in elements that are used within Revit.

6.1. Externalizability

State is messy in Dynamo. The computational graph carries state that is typically associated with the execution of a program. However, many of the variables in the graph are really pointers to objects in the "world" of Revit. This means that the state is shared between the Dynamo virtual machine and the Revit document model. Approximately the intent is that the program is represented within the Dynamo document, the state in the Revit document, and the Dynamo "document" also stores pointers as to how to link the two together. This aligns to the design agenda that the Dynamo project embodies, that it is a mediator of distributed "cognition" amongst computational modes of representation (Dynamo), architectural ones (Revit) and the unknowable representation in the users. However as this structure is reified into software architecture, the boundaries get messy.

6.2. Actuals and Possibles

The separation of the software architecture along the lines describes above, whilst making accounting for state complicated, fairly naturally enforces the separation between Res Potentia I and Res Potentia II. The definition of what is possible is in two places: the core data types in the virtual machine (numbers, strings, maps), and the more complicated object types that contain things useful to architects (points, lines, planes, doors, walls, chairs). These are defined on the target side of a Foreign Function Interface boundary. In Dynamo's case this is a C# wrapper over the Revit API or a geometry kernel. These makes these definitions technically and socially separated from the nodes that use them. This strategy has been sufficiently successful that the project has progressively moved towards eliminating the possibility of defining objects on the Dynamo VM side. Res Potentia II, using the objects, on the other hand, is all on the Dynamo VM side, with the construction calls corresponding roughly to nodes in the graph.

The Extensa of a system employing this distributed style of programming where the state is shared across many technical boundaries is much more challenging to design. The Extensa represents a totality of the state across the ecosystem, including the representation of objects that are only partially computationally accessible (not everything in Revit has an API that Dynamo can use). This represents a substantial ongoing challenge in programming language design: how to design systems that do not expect to have a totalist perspective over the worlds they manipulate, or even over the potentia for those worlds.

6.3. Divergence

With a system that embodies this distributed Potentia and lack of total knowledge, divergence ceases to be only an inconvenience and becomes a primary usability problem. In Dynamo we address this in a couple of ways. The virtual machine employs an indirection layer, names are bound to transitory "shells": these can dynamically change which objects they actually point to. These shells are computationally

[3] What You See Is What I See

```
function makeThing () {
    var thing = fluid.thing({arg: "value"});
    ...
}
```

Listing 1 – A highly divergent Infusion usage

efficient, but also allow a decoupling of the virtual machine's view of identity.

This allows us to employ a Queen of Sheba identity scheme. On each partial execution, we generate a "trace" of new shells associated with the call sites of the program. We then model the structural correspondence in the layout of these shells compared to the previous trace. Heuristics such as comparing the cardinality of the arrays are used to determine if they structurally match. If they do, the shells adopt the identity of the previous objects. If the correspondence can't be matched, new objects are created for the shells, and any that don't have remaining shells are garbage collected. This strategy has been reasonably successful at managing divergence in a way that isn't too surprising to users using the system — however it remains to seen how successful it will be as the computational complexity of the graphs being manipulated grows.

7. Infusion

Infusion currently takes the form of a configuration dialect expressed in terms of JSON structures, organising the activity of short, publicly named, mostly side-effect-free functions written in JavaScript. Rather than considering Infusion specifically as either a language or a system (see (Gabriel, 2012) for the interesting history of this distinction) we prefer to consider it as an *integration domain*, following the thought of (Kell, 2009).

Much of the design of Infusion has been motivated by the minimisation of divergence, although this term had not been coined for most of its history. Infusion draws up a taxonomy of application function, for example events and their listeners, model material and relationships between it, that allows this function to be authored in terms of documents (following the sense of section 2).

The *Nexus* (The GPII Team, 2016) is an experimental system which fully externalises the affordances of a running Infusion system in terms of simple JSON payloads exchanged over HTTP and WebSockets protocols. To the extent that the Nexus is functional, it represents a system in which does indeed fully separate the three areas of expression — however, it has many implementation gaps that need to be filled before it is ready for purposes beyond simple technology demonstrations.

7.1. Potentia I & II in Infusion

Infusion does not yet clearly allow the user to separate potentia II expressions from their extensa, but this feature is close enough to being real that we can helpfully illustrate what this distinction is in terms of Infusion features.

Listing 1 shows a traditional programming-language use of Infusion to construct an object, which illustrates several problems. In this snippet, both the *intention* (to create an instance of type `fluid.thing`) as well as the *result* (the instance itself) are private: the fact that `fluid.thing` has been requested is knowable only to the runtime at that moment (or else some highly sophisticated tool capable of parsing arbitrary JavaScript and matching it against a complex schema of possible function effects), and the result is stored in a inaccessible area — a function scope, from which it is not accessible by any code elsewhere in the system, and is not even accessible by traditional debugging tools unless the stack frame for the function happens to be activated at that moment.

A somewhat less divergent incarnation of this expression is shown in listing 2. The first argument to `fluid.construct` holds the *global address* of the component to be constructed, expressed as a set

```
fluid.construct(["thing"], {
    type: "fluid.thing",
    arg: "value"
});

```

Listing 2 – Potentia II expression with reduced divergence from that of listing 1

```
HTTP POST /components/thing {"type": "fluid.thing", "arg": "value"}
```

Listing 3 – Externalised form of the expression shown in listings 1 and 2 as a Nexus HTTP request

of path segments[4]. The second argument holds the description of the component, expressed in an easily serialisable JSON-equivalent form. This expression *could* be treated as simply a specification of the potentia II material designating the instantiation of the component, and form a separate workflow to actually initiating the instantiation itself. Unfortunately in the current framework, this function call is treated as a directive to synchronously instantiate the component.

To complete the externalization picture, listing 3 shows the same expression as issued from outside the system via the HTTP Nexus API. Here we are tantalisingly close to having the potentia II expression separable from the executing system itself. All we would need to do is keep a record of such requests incoming into the system in a readily indexable form, or perhaps place an HTTP proxy layer in front of the system which achieve this. Such things have so far not been done.

7.2. Stack divergence in Infusion

Infusion currently makes no efforts to fight divergence from the stack. Since our tooling is quite immature, we rely heavily on standard debugging tools in which the presence of a long and detailed stack trace is essential in order to understand the action of the system. An upcoming rewrite will improve the system's treatment of asynchronous resolution, which will result in loss of quality in the authorial experience described in section 4.1 unless it can somehow be compensated by development of powerful tools to help visualise the trajectory of the system.

7.3. Reducing listener divergence in Infusion

Infusion successfully eliminates divergence resulting from listener definition and registration. Listeners definitions are attached to component definitions, and have their lifetime automatically scoped to the lifetime of the component holding the definition (which may be different to the component to which the listener is attached). Since each definition is assigned a unique address within the system's space of definitions, the identity of the listener is stable (with respect to the potentia), despite being honoured by different function handles in the running system from time to time (the extensa). In addition, listener definitions can be manually assigned a *namespace* which enables them to be targeted by further authorial expressions — that is, a particular listener definition can be overridden by supplying its namespace when supplying a further definition which already matches both in target component and event name. Listing 4 shows a simple setup with three components, A, B and C where construction of B will register a listener attaching one of C's methods to one of A's events. Destroying B will tear down this listener, and reconstructing B would regenerate it, in accordance with the "Queen of Sheba" dynamics described in section 4.4.4. Since at present Infusion has no transaction system governing overall component instantiation (it has transactions governing only the *model skeleton* attached to components), this destruction and recreation would be sadly evident to all.

[4]Note that this externalization of component addresses raises its own hazards. The privacy/opacity of traditional addresses served a useful authorial purpose — it was impossible to "accidentally" (or indeed at all) overwrite one author's component with another. The shared inhabitation of a global address space requires careful management of a hierarchical space of names — just as in similar shared spaces as the DOM. This is to be done via conventions allocating well-known component root addresses to different authors — by schemes similar to those used to avoid collisions in package names in global namespaces such as the Java package system.

```
fluid.defaults("examples.ppigListeners", {
    gradeNames: "fluid.component",
    components: {
        A: {
            type: "fluid.component",
            options: {
                events: {
                    fireIt: null
                }
            }
        },
        B: {
            type: "fluid.component",
            options: {
                listeners: {
                    "{ppigListeners}.A.events.fireIt": "{ppigListeners}.C.hearIt"
                }
            }
        },
        C: {
            type: "fluid.component",
            options: {
                invokers: {
                    hearIt: "fluid.log"
                }
            }
        }
    }
});

// Sample usage:
// Construct overall tree
var that = examples.ppigListeners();
// causes logged message "I send it" via C's method mediated by B's binding
that.A.fireIt.fire("I send it");
// destroy B, thus tearing down its binding
that.B.destroy();
// No effect since event binding has been removed
that.A.fireIt.fire("I send it again");
// Note that B cannot in this design be easily recreated since we do not have separated potentia II records
```

Listing 4 – Declarative registration of divergence-free listeners in Infusion

8. Conclusion

We have critiqued the values of live programming, showing that many of their aims can be met without necessarily adhering to the letter of liveness, and that they can be situated amongst wider considerations of authorship for software systems in general, supporting cycles of authorship and use by networks of authors working in different times and places. We have introduced a quantity named *divergence*, a measure of the discrepancy between the internal bookkeeping state required to maintain a running, live system, and the state by which it can be authored by means of externalizable documents. We have linked the authorial cycles that we wish to support both to the system of Potentials and Actuals envisioned by Whitehead in his *Process and Reality*, as well as to the system of maintenance of the universe described by Ibn al-Arabi in his *The Bezels of Wisdom*.

We have considered three real systems (Webstrates, Dynamo and Infusion) from this point of view, and shown how much work still remains before a substantial fraction of the aims of open authorship can be achieved.

9. References

Al-Arabi, I. (c. 1229). *Fusus Al-Hikam ("The Bezels of Wisdom")* (Binyamin Abrahamov, Ed.). Routledge, 2015. Retrieved from `http://bewley.virtualave.net/fusus16.html`

Black et al. (2011). The Smalltalk Object Model. In *Pharo By Example.* Retrieved from `http://pharo.gforge.inria.fr/PBE1/PBE1ch6.html`

Bryant, A., & Putney, C. (2016). *Monticello.* Retrieved from http://www.wiresong.ca/monticello/

Church, L., Söderberg, E., Bracha, G., & Tanimoto, S. (2016). Liveness becomes Entelechy - a scheme for L6. In *The second international conference on live coding.*

Fielding, R. T., & Taylor, R. N. (2000). Principled design of the modern web architecture. In *Proceedings of the 22nd international conference on software engineering* (pp. 407–416). New York, NY, USA: ACM.

Gabriel, R. P. (2012). The structure of a programming language revolution. In *Proceedings of the ACM Onward 2012* (pp. 195–214).

Kauffman, S. (2011). *A hypothesis: Res potentia and res extensa linked by measurement.* Retrieved from http://www.npr.org/sections/13.7/2011/01/03/132607500/an-hypothesis-res-potentia-and-res-extensa-linked-by-measurement

Kay, A., & Goldberg, A. (1977). Personal dynamic media. *Computer*, *10*(3), 31–41.

Kell, S. (2009). The mythical matched modules: overcoming the tyranny of inflexible software construction. In *Proceedings of the 2009 oopsla companion (onward)* (pp. 881–888).

Klokmose, C. N., Eagan, J. R., Baader, S., Mackay, W., & Beaudouin-Lafon, M. (2015). Webstrates: Shareable dynamic media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology* (pp. 280–290).

Knuth, D. E. (1992). *Literate programming*. Stanford University Center for the Study of Language and Information.

Kulkarni, M., Carribault, P., Pingali, K., Ramanarayanan, G., Walter, B., Bala, K., & Chew, L. (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.*

Pharo. (2016). *Pharo: The Immersive Programming Experience.* Retrieved from http://pharo.org/

Squeak Project. (2016). *Squeak/Smalltalk.* Retrieved from http://squeak.org/

Tanimoto, S. L. (2013). A perspective on the evolution of live programming. In *Proceedings of the 1st international workshop on live programming* (pp. 31–34). Piscataway, NJ, USA: IEEE Press.

The GPII Team. (2016). *The GPII Nexus.* Retrieved from https://wiki.gpii.net/w/The_Nexus

Ungar, D., Chambers, C., Chang, B.-W., & Hoelzle, U. (1991). Organizing Programs Without Classes. *Lisp and Symbolic Computation*, *4*(3). Retrieved from http://bibliography.selflanguage.org/_static/organizing-programs.pdf

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

Whitehead, A. N. (1929). *Process and Reality*. Free Press.