Tracing a Paradigm for Externalization: Avatars and the GPII Nexus

COLIN CLARK, OCAD University
ANTRANIG BASMAN, Raising the Floor

We will situate the concept of an **avatar** *(a working simulacrum of part of a system separated from it in space or time) with respect to traditional concepts of programming language and systems design. Whilst much theory and practice argues in favour of* **insulation** *(the creation of architectural boundaries prohibiting the leakage of information) we will find that many successful systems take a diametrically opposed approach. We name this family of systems as those based on* **externalised state transfer**. *Rather than hiding implementation details behind APIs, object interfaces or similar, these systems actively advertise their internal structure and its coordinates via data and metadata. Examples of these systems include RESTful web applications, MIDI devices, and the DWARF debugging file format. We discuss such systems and how we can purposefully design new systems embodying such virtues in a more distilled form.*

1 AVATARS

An avatar is a part of a system of which a working simulacrum can be naturally embodied elsewhere. The simulacrum is a functioning replica of the remote system, which can stand in for it to some level of faithfulness without having to ask questions of it at each interaction. The notion of "naturalness" we hope to establish, however, cannot be determined by examining properties of a single group of interacting systems. Instead, we will need to consider ecologies of such related systems, their users, and networks of other authors who work with them. By exhibiting different kinds of systems which we consider meet and do not meet our definition, we hope to sharpen understanding of the distinction we are interested in, as well as promote new development techniques that make the implementation of avatars natural and idiomatic.

We have borrowed and extended the term "avatar" from its traditional use describing a visual or schematised representation of a user of a system. In our usage, an avatar is a more or less symmetrical concept: two systems may be avatars of one another if they meet the criterion of faithfulness or representation without one of them being necessarily designated as "primary" or "real".

The most common appearance of the avatar pattern is in supporting a remote user interface, whereby a single system exposes the potential for control to different locations, separated by a network of some kind. To the extent that the remote system might be agnostic as to whether the controlled system is local or remote, it could be said to be working with an avatar, or functioning local replica of the controlled system. However, whilst it is one of the most convenient, the avatar pattern is an uncommon choice for implementing such a remote UI. A more common pattern has the remote UI bound to the controlled system by a message bus along which passes essentially arbitrary messages — that is, messages with individual schemas whose content do not correspond to an integrated view of the system state. We could describe this pattern as a remote procedure call (RPC) or API idiom where the message bus simply stands in for whatever function calls the two parts of the architecture found it necessary to make to each other from time to time. Examples of this remoting pattern are the X Windows protocol, the Microsoft Windows Remote Desktop protocol, Virtual Network Computing, etc.

1.1 Avatars for the future

There are numerous other productive uses of the avatar model, which need not be limited to scenarios where the separation is in space. The avatar model could be used to support advanced live programming scenarios [3], where a system that is being authored can be asked, explicitly or implicitly, to speculate about states it might reach in the future as a result of different authorial gestures. This supports the highest level of liveness (L6) in Church et al. [4]'s classification of live programming paradigms. In this case, part of the current system becomes an avatar for part of its state in the future — it is a working simulacrum allowing the user to see some or all of its behaviour, but which is integral in itself and may be immediately discarded without any side effects on the behaviour of the current system.

1.2 Avatars from REST

However, it's not accidental that by far the most prolific technology for supporting remote user interfaces, the web, is also the birthplace of a software idiom, REST [6], which is a foundational part of the avatar model. In REST, "representations of state are transferred" — that is, the remote system does not simply answer arbitrary messages, but instead guarantees to transfer complete (exhaustive) representations of part of its application state to the UI client. The expectation is that the client then operates on this state in the form of a local avatar of some greater or lesser degree of faithfulness, as a functioning replica of some aspect of the remote system. The extension that the avatar model represents with respect to REST is that we provide for transfer of representations capable of dynamic evolution of behaviour. This distinction is elaborated in Basman et al. [3] where we describe the extension that a *potentia* represents with respect to a system, like the web, with the document object model (DOM) for its pages transmitted over HTTP as HTML, which only provides for representing an *extensa*.

1.3 Avatars not from objects

There are descriptions of systems which appear to conform to our description of an avatar, but which we recognise as part of a distinct conception. Bank [2] contains a now legendary interview by Wired Magazine with James Gosling, in which he describes an experience listening to a concert, watching "semirobotic lights which seemed to dance to the music" and being inspired by them to think differently about "making behaviour flow through networks". The resulting language that he designed, Java, is a paradigm member of what we named the API idiom for distributed behaviour. But there is a subtlety to be worked out here — since Java, Smalltalk *et al* do indeed allow behaviour to be shipped wholesale around the network and reconstituted at a remote site for further execution. Why do

we not admit such a system as constituting a usage of the avatar pattern we are describing?

It is because the shipped representation is necessarily opaque to any consumers except for the runtime of the remote system, and that (and hence) the representation itself supplies no assistance as to how to establish conformance in representational state between the systems it is shipped between. A Java applet rendering a user interface served from a Java server can derive very little assistance from the conformance in languages and virtual machines, even if it makes use of Java's serialisation facilities in shipping complete objects (that is, combined state and behaviour) around the network. In practice, real-world Java applications communicate at runtime via standard, arbitrary APIs. This is because they are expressed in terms of "objects", language and runtime artefacts expressed in a paradigm where information hiding and insulation are seen as virtues. In the paradigm we are proposing, they are the reverse — a reversal we'll examine in the next section.

1.4 Avatars against information hiding

Almost all current programming languages, their philosophies (whether procedural, functional, or object-oriented), toolchains and implementation technologies (compilers, virtual machines) work to obstruct the efficient creation of avatars. We assert that the typical tendency of thought by software engineers and computer scientists is to only cater for cases where a system is operated on from inside itself, neglecting its participation in wider authorial networks where it is embedded in wider contexts of use, by clients separated in space and time, using different languages, idioms and technologies.

Sometimes the neglect of these wider contexts of use is active and intentional. A significant lineage in favour of information hiding can be traced back to Parnas [8]. Here, Parnas' primary motivation for information hiding was to establish more efficient, centralized management processes that would help scale software development practises for increasingly large, disconnected teams of programmers of varying skill levels. Unfortunately, as is often the case with foundational developments in a nascent field, the advice is taken on board, and the historical and social context in which it arose is forgotten. Parnas' social structure for software development teams became, over time, sedimented into the technical infrastructure of programming languages and design philosophies. Yet today's development is equally as likely to occur on an individual or local scale as against a corporate backdrop, and fundamentally different considerations than those which originally embedded information hiding as a core virtue underlying programming language development may well be appropriate.

In practice, whenever artefacts are embedded in a particular context of use where avatar-style working is strongly required, certain technological patterns have emerged, often informally, which support this style of working. These patterns, involving the use of externalised state as we discussed in the previous two sections, are either overlooked or if they are observed, are usually decried by traditional software technologists as being antithetical to their values ("The web is broken!", as seen in Tiselice [12] and others). These patterns support the avatar model by purposing the messages which are transported between systems as "state transfer" in the REST sense. The fact that these messages are supported have implications for how the remote architecture is seen by others, and usually also on how it sees itself — rather than lying in opaque "black boxes" beloved of the proponents of information hiding, object orientation, or the API model, the architecture is instead coordinatised [3], with its elements laid out in a spatial grid or tree, each element of which has well-known coordinates and whose state is in theory available for inspection and modification at any time.

2 MIDI DEVICES

An early, partial example of externalized state transfer is MIDI (Musical Instrument Digital Interface). Instruments and controllers equipped with MIDI, such as synthesizers, drum machines, sequencers, and patch editors, have proliferated over the last 35 years. Indeed, MIDI has had an unusual longevity: despite appearing antiquated by today's standards, it remains one of the most widely-supported interoperability technologies available, and most new musical devices today still come equipped with it. For example, the Yamaha DX7, one of the first MIDI synthesizers available, was released in 1983; Patch Base, an iOS patch editor used to "program" the DX7, was updated by its developer only days ago. Musicians still regularly employ original, decades-old MIDI hardware, and can integrate it with new software and platforms. What gives MIDI devices this longevity, which seems so unlike today's rapidly-obsolescing general-purpose software products?

2.1 System Exclusive Messages in MIDI

One of the reasons why many old MIDI instruments continue to be musically viable is due, perhaps counterintuitively, to its least "designed" aspect: system exclusive (SysEx) messages. The content of SysEx messages was never standardized by the Music Manufacturer's Association (MMA); system exclusive messages are free, vendor-specific messages that can, in theory, convey any kind of information to or from a MIDI device. However, in practice, MIDI SysEx messages were immediately used in a semi-conventionalized way as a means for externalizing the complete state of a musical device – all its patches, voice parameters, and settings. This was undoubtedly motivated by the storage constraints of the era; in Smith and Wood [10], MIDI's designers only anticipate SysEx's use as a means for loading and saving patches to and from external storage. In practice, however, this comprehensive externalization enabled an unexpected ecosystem of third-party, software-based patch editors and alternative control hardware to emerge. Most MIDI manufacturers provide a comprehensive (if not standardized) means for remote applications and devices to query and modify all aspects of the device's state, both in whole (an entire map of patches) or in part (a single parameter value for one voice). As a result, completely novel user interfaces have been designed to make programming the complex, single-line menu interfaces of old synthesizers substantially easier. Similarly, patch "morphers" and randomizers have been developed to algorithmically assist composers in generating novel sounds on these devices. It is this ecosystem of unanticipated remote programmability that has extended the value of old MIDI devices well beyond their designer's expectations, allowing them to adapt to today's increased computing power, greater user interface sophistication, and the rise of mobile devices.

2.2 Avatars for the physical

A patch editor's user interface can be seen as an avatar for the ancient synthesizer, controlling it over a stream of MIDI messages whose format has not changed over this timespan. Accidental though it was, MIDI devices have a key quality that is divergent from most modern software architectures: they provide a fully externalized view of the entire device's state, and this state can be operated on, in whole or in part, externally.

MIDI's view of state is, by today's standard, unquestionably anachronistic and far too low-level, yet nonetheless offers a view into a "characteristic of the machine". The SysEx protocol allows for the exchange of essentially arbitrary streams of bytes, and offers the barest formality for framing these byte streams and identifying their sources. One might expect this lack of constraint to result in arbitrary, RPC-like idioms for message encoding and interpretation. Instead, the dominant idiom for these messages is to simply transfer certain sections of the device's memory to and from the host. Contrary to the proponents of reuse who promote insulation and information hiding as the primary routes to this goal, devices based on this idiom have proved the most durable and reusable in historical practice.

2.3 Falling away from the avatar pattern

Along the way, however, it seems that this architectural benefit is increasingly lost to developers. For example, iOS is becoming increasingly popular as a platform to both develop new musical instruments as well as to emulate older hardware devices. Yet few, if any, iOS synthesizers provide any means for externalizing their state, via SysEx or otherwise. The only way to edit patches or customize their state is via their graphical user interface. Even applications such as the Roland Sound Canvas app, which claims to "emulate perfectly" its popular hardware namesake from the 1990s, specifically omits the SysEx implementation of the original Roland GS specification. This prevents it from being in any way externally edited, customized programmatically, or used via an different, context-appropriate UI. Here, in the realm of sophisticated and polished user interfaces, there is no longer a means to alternatively control, present, or configure the virtual device. The designer's plan is absolute.

2.4 MIDI's false future: OSC

Newer protocols such as Open Sound Control (OSC), which some have suggested could eventually replace MIDI, address some of the anachronistic qualities of MIDI (such as its 7-bit value resolution), but do little to promote the more sophisticated forms of externalization that we outlined in the previous section. In particular, OSC lacks any way to express state within the coordinatized hierarchy of a "document" that formats such as JSON provide. As such, OSC actually makes it significantly more difficult to transfer full, faithful avatars of a system's state while retaining addressability; state typically needs to be transferred one property at a time. In practice, nearly all remote interfaces implemented using OSC tend towards the RPC-like idiom, providing only a way to remotely invoke functions. This makes it difficult to support the serendipitious or unanticipated use of a system without the consent and intervention of its designers.

3 UNIX PROCESS METADATA

Kell [7] observes that, through the practical necessity of supporting the crucial authorial activity of debugging, UNIX processes have *de facto* been supplied with sufficient metadata to support substantial introspection into their allocation patterns, which can be recovered from the DWARF metadata accompanying object code and other sources. This is in contrast to the typical computer science design recommendations of providing *services* to support such faculties of reflection. The traditional VM approach to reflection has "the reflecting client consume the services of an in-VM reflection API and/or debug server". [7] notes that such an approach would be substantially limited in function (not supporting the post-mortem case of debugging against a "dead" core dump) and portability (not supporting the use of one vendor's debugger to debug code from another's compiler). By casting its task in terms of externalising access to the state of the system, by mapping its addresses in terms of metadata, the process model promotes the capability of parts of the system to act as avatars for others, even to the extent of bridging the divide between the living and the dead (being able to treat in-memory, running processes on common terms to core dumps in files). Similarly to the ability of state transfer-based MIDI devices to enjoy huge longevity and portability, Kell [7] notes that for a system cast in terms of an API, "it becomes hard to implement reflection features not anticipated in the design of the reflection API or debug server command language. By contrast, metadata is open-ended and naturally decoupling".

4 THE NEXUS

The GPII Nexus [5] is a concrete implementation of the values and idioms of the avatar model. It provides a means for connecting together software components that may have been implemented using different programming languages, toolkits, and frameworks, which may be running on different devices or processes. The Nexus is being developed as part of the Prosperity4All Project [9], a European Commission-funded project that aims to reduce the cost and complexity of building assistive technologies and adaptive user interfaces.

4.1 Current Nexus Architecture and Integrations

The Nexus is currently implemented as a JavaScript application written in Node.js, which exports a fully addressed tree of implementation cells, each named a component, over widely supported public web protocols such as HTTP and WebSockets, with payloads encoded as JSON [1]. The underlying substrate for the Nexus is an in-memory tree of components managed by the Infusion framework [11] where the exported JSON payloads correspond to Infusion's declarative dialect for encoding application function.

Clients of the Nexus can be easily implemented in any language or platform, including within web browsers, low-powered devices, and mobile or desktop platforms. So far, we have developed clients in both Java and JavaScript using Web Sockets, and have deployed Nexus clients on embedded platforms including the Raspberry Pi

```
HTTP PUT /defaults/examples.minimalGrade {gradeNames: "fluid.component"}
HTTP POST /components/minimalInstance {type: "examples.minimalGrade"}
HTTP DELETE /components/minimalInstance
```

Listing 1. Some simple Nexus directives encoded over the HTTP protocol

and C.H.I.P. board computers. Notably, we have recently used the Nexus to develop a new distributed musical instrument that provides custom user interfaces that help to engage a diversity of users in performing music. We have designed Nexus-connected UIs that provide different ways to collectively improvise music using alternative keyboards, note grids, mobile accelerometers, and head-trackers. The goal of this project is enable non-musicians and people with disabilities to participate fully in the creative process.

4.2 Examples of the Nexus API

When addressing a particular component in the tree, a segment of the Nexus API corresponds to the well-known "CRUD over REST" protocol whereby resources are managed by HTTP verbs at a particular URL. Examples of some very simple Nexus directives are shown in Listing 1, where we register a new grade (potentia I element), create a component of that grade at the root of the component tree (using a potentia II directive) and then destroy the component.

However, when taken as a whole, the Nexus API supports the transmission of wholesale avatars. Querying the contents of a particular section of the Nexus' component tree over HTTP allows it to be completely replicated at another site, together with any dynamic behaviour.

4.3 Encoding dynamic behaviour

The dynamic behaviour of a Nexus (Infusion) application includes the capacity of the system to create or destroy further components, The distinction between systems which close over this capacity and those which do not is an important one, analogous to our subsumption of REST in section 1.2. The web technologies which we describe as the inspiration and foundation for Infusion are missing this capability, since they can only close over the DOM, which at any time simply expresses the capabilities of a static document. In the terminology of Basman et al. [3], the DOM simply constitutes an *extensa* whereas a fully capable dynamic system must also encode state corresponding to two kinds of *potentia*. If, for example, clicking on a button causes a new application region to be created, this capability must necessarily be encoded outside the DOM — since if it was encoded within the DOM, this leads to the absurd result that every possible application condition is encoded somewhere within a gigantic, extensive DOM — this absurdity is closely related to the ancient fallacy of *preformationism* [13] in which the reproduction of organisms was believed to be enabled by every possible recursive descendent of an organism being enclosed within it in miniature form.

Encoding the two kinds of potentia — *potentia I*, loosely analogous to types or classes in traditional languages, and *potentia II*, the aligned registry of user expressions, which has no equivalent in traditional languages — has been one of the principal tasks of Infusion development over the last few years. Interested readers can consult Basman et al. [3] and the Infusion framework documentation and related JIRA tickets for details.

4.4 Requirement for transactional updates

A crucial element of any successful avatar system is transaction demarcation. It must be possible for a sequence of related Nexus creation or destruction messages (in the terminology of Basman et al. [3], such messages are *potentia II elements*) to be grouped as part of a transactional unit which must be honoured completely or not at all, in order to ensure that the system is left in a state meaningful to users. This capability is implicit in our definition of an avatar system: since these are separated in space or time from one another, the fidelity with which one may stand in for another must be resolved with respect to some kind of minimum granular unit.

It is an important goal of the Nexus to formalise such demarcation models as well as providing infrastructure for supporting reference implementations. Transactions are even more essential within avatar systems than they are in traditional database applications storing "dead data" over CRUD - since the moment an avatar is transferred, its state may immediately start to evolve. It's essential that there can be a model of atomicity for these updates so that partially transferred avatars do not begin to corrupt their state. This atomicity is also crucial to support "avatars over time," which we sketched in section 1. If we begin to speculatively execute some possible future configuration of the system, it is essential that it can be completely backed out without effect if it is not desired in the present.

To support this, the Nexus will support a model inspired by current generations of distributed version management tools such as Git, which assign globally stable hashes to the configuration of parts of its tree, to which the tree could always be restored assuming that the storage backing the hash has not been discarded. Every client communication is enlisted in such transactions, in which they are operating on an avatar that is isolated from those visible by other clients, until they explicitly commit their transaction. At this point, the updates to the avatar will be resolved. If they cause an error either with respect to the internal avatar contents, or with respect to the rest of the tree, the entire transaction will be backed out and the communication ended without any externally visible side-effects. This process allows safe experimentation with arbitrarily complex updates to application state, many of which could be speculatively executed against local or remote avatars, supporting the use cases of L6 programming in the sense of Church et al. [4]. The choice of where to execute the update can be driven by the economics of the situation — the relative compute power of the client and its peer, network costs, latency requirements — rather than being a choice forced by the software architecture as is almost always currently the case.

5 CONCLUSION

We've described the conception of the **avatar** and shown its relationship to the externalised state transfer idiom, which is one that arises naturally in communities which need to produce artefacts that are shared as part of an open ecology of function. The goal of the externalised state transfer idiom is to encourage and facilitate

the implementation of avatars, resulting in harmonious, democratic and open experiences of authorship by integrated communities, rather than proceeding in a hierarchy of command from a technical elite. The pleasant results of this idiom include live programming systems able to assist the user to speculate about the future effects of current authorial decisions, systems which expose a rich variety of interaction idioms and technologies suited to a diverse community of authors and users, and systems which enjoy huge longevity and support through ditching reliance on brittle idioms based on contracts, function calls, and encapsulation boundaries.

6 ACKNOWLEDGEMENTS

The authors would like to thank Simon Bates, the lead developer of the Nexus, for his contributions to this paper and to the Nexus as a whole. This research, part of the Prosperity4All Project, was funded by the European Union's Seventh Framework Programme (FP7/2007-2013) grant agreement no. 610510.

REFERENCES

[1] Antranig Basman and Simon Bates and Colin Clark. 2016. Nexus API. (2016). https://wiki.gpii.net/w/Nexus_API

[2] David Bank. 1995. The Java Saga. (1995). https://www.wired.com/1995/12/java-saga/

[3] A. Basman, L. Church, C. Klokmose, and C. Clark. 2016. Software and How it Lives On - Embedding Live Programs in the World Around Them. In *Proceedings of the Psychology of Programming Interest Group*.

[4] L. Church, E. Söderberg, G. Bracha, and S. Tanimoto. 2016. Liveness becomes Entelechy - a scheme for L6. In *The Second International Conference on Live Coding*.

[5] Colin Clark and Antranig Basman and Simon Bates. 2016. The GPII Nexus. (2016). https://wiki.gpii.net/w/The_Nexus

[6] Roy T. Fielding and Richard N. Taylor. 2000. Principled Design of the Modern Web Architecture. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*. ACM, New York, NY, USA, 407–416.

[7] S. Kell. 2015. Towards a Dynamic Object Model within UNIX Processes. In *Proceedings of the 2015 OOPSLA Companion (Onward)*. 224–239.

[8] D.L. Parnas. 1971. Information Distribution Aspects of Design Methodology. *Methods* 4, 5 (1971), 6–7.

[9] Matthias Peissner, Gregg C. Vanderheiden, Jutta Treviranus, and Gianna Tsakou. 2014. Prosperity4All–Setting the Stage for a Paradigm Shift in eInclusion. In *International Conference on Universal Access in Human-Computer Interaction*. Springer, 443–452.

[10] Dave Smith and Chet Wood. 1981. The USI, or Universal Synthesizer Interface. In *Audio Engineering Society Convention 70*. http://www.aes.org/e-lib/browse.cfm?elib=11909

[11] Fluid Team. 2017. Fluid Infusion Documentation. (2017). http://docs.fluidproject.org/infusion/development/

[12] Dragos Tiselice. 2015. Web sucks and here's how we can make it awesome. (2015). https://www.presslabs.com/blog/web-sucks-how-to-make-it-awesome/

[13] Wikipedia. 2017. Preformationism. (2017). https://en.wikipedia.org/wiki/Preformationism