An Anatomy of Interaction: Co-occurrences and Entanglements

Antranig Basman
Raising the Floor - International
London, England
amb26@ponder.org.uk

Philip Tchernavskij
LRI, Université Paris-Sud, CNRS,
Inria, Universitè Paris-Saclay
Orsay, France
ptcher@lri.fr

Simon Bates
OCAD University
Toronto, Canada
sbates@ocadu.ca

Michel Beaudouin-Lafon
LRI, Université Paris-Sud, CNRS,
Inria, Université Paris-Saclay
Orsay, France
mbl@lri.fr

ABSTRACT

We present a new taxonomy for describing the conditions and implementation of interactions. Current mechanisms for embedding interaction in software promote a hard separation between the programmers who produce the software, and the communities who go on to use it. In order to support open ecologies of function and fabrication, where this separation is negotiated by communities of users and designers, we need to reconceive those mechanisms. We describe interaction in two phases: Co-occurrence, the prerequisite conditions for an interaction to take place; and entanglement, the temporary coupling and interplay between elements participating in the interaction. We then sketch a blueprint of a system where those phases and their adjacent mechanisms enable communities of users to build and use interactive software. There are many ways of conceiving this new design space, and we present three dominant metaphors which we have employed so far, based on chemical reactions, quantum physics and cooking. We exhibit different systems which we have implemented based on these metaphors, and sketch how future systems will further empower citizens to design and inhabit their own interactions, express ownership over them and share them with communities of interest.

CCS CONCEPTS

• **Human-centered computing → Interaction paradigms**; **Interactive systems and tools**;

KEYWORDS

Interaction models, interaction paradigms, interaction architectures

ACM Reference Format:
Antranig Basman, Philip Tchernavskij, Simon Bates, and Michel Beaudouin-Lafon. 2018. An Anatomy of Interaction: Co-occurrences and Entanglements. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (*Author version *<Programming'18> Companion)*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3191697.3214328

Author version
<Programming'18> Companion, April 9–12, 2018, Nice, France
2018. ACM ISBN 978-1-4503-5513-1/18/04…$15.00
https://doi.org/10.1145/3191697.3214328

1 INTRODUCTION

To support human activities, what computers do must be about more than just computation. In a reframing of computational activity, Wegner [20] argues that "Interaction is more powerful than Algorithms". We support Wegner's problematisation of computation, that "Algorithms and Turing Machines, like Cartesian thinkers, shut out the world during the process of problem solving". Taking computers and computation beyond the role of "problem solvers", we want to embed their activities in the real world as part of an *open ecology of function* [9]. We want to facilitate and empower creative networks to curate and share artefacts of interest. To reach this goal, we need to tease apart the nature of what interaction **is**, what it relies upon, and how its prerequisites can be organised and filtered in order to allow interactions themselves to take the role of first-class artefacts of interest. In this paper, we reveal substructure in what has been traditionally treated as an atomic phenomenon. We begin by sketching our goals for software artefacts within communities of interest, then characterise what we are prepared to recognise as an interaction, and then consider how these goals influence the structure of what we recognise.

2 ECOLOGIES OF FUNCTION AND FABRICATION

Basman and Clark [2, 9] introduce the notion that artefacts could and should take part in open ecologies. The ecology of **use** or **function** reflects how an artefact comes to be recognised and used amongst a collection of similar artefacts with similar functions — for example, the ecology of spoons presents obvious and related affordances based on their relative size, shape and materials.

The ecology of **construction** or **fabrication** establishes relationships amongst creators which may or may not mirror those amongst the artefacts which are being constructed — different craftsmen may specialise in spoons of different materials, different dimensions or context of use. These ecologies can be more or less open to the extent they enable new participants to enter or transition into the ecology, as users or fabricators. Expert users may have the option to make connections to relevant members of the fabrication ecology, if it is sufficiently open, and perhaps start taking on some of their skills. However, what is good for spoons is not good for software — even the most basic of the skills necessary to construct or modify acceptable software will remain out of reach of the vast majority of citizenry who could benefit from their use, because the relevant ecologies are closed.

2.1 Open Authorship

So-called end-user development (EUP) [17] has been a goal of many communities over decades, but we argue that an important impediment to useful results has been a failure to concentrate on the material qualities of software that influence the structure of the above ecologies [2]. Our work in drawing up and materialising our anatomy of interaction will assist the goals of EUP only indirectly — since as the literature shows, a head-on assault on this profound problem will not lead to generally useful results. We see our anatomy as a piece of cognitive and implementational scaffolding on which future EUP systems can be productively built; the code and artefacts presented in this paper will not be manipulated by users of the system directly, but are planned to allow the expression of a future superstructure of tooling to be more harmonious and amenable.

The form of current software changes radically between the two ecologies, from brittle and largely textual code to opaque and largely visual interfaces. Code is changeable by those with the tools and status of the fabrication ecology, but once it is in use, a piece of software cannot generally return to a changeable form. Whilst there are social and economic drivers of this separation, Basman et al. [3] describe one technological driver as the problem of *divergence* between dead and live forms of software.

The purpose of bringing these forms into convergence is to support the "open authorial principle" described in [4] — that any expression by any one author of a system can have its effect replaced via an *addition* to the design by another author.

Some previous work has approached the goals of open authorship: For example, in the *Buttons* project [16], a community of users tailored their operating systems as part of normal use by creating, customising, and sharing buttons containing Lisp scripts. Notably, MacLean et al. emphasise the parallel project of creating a tailoring toolkit side-by-side with a tailoring culture. The toolkit allowed users to progressively unfold and edit buttons, accessing their representation, then command parameters, then the functions that define their behaviour. The culture consisted of relationships among non-programmers and programmers who collaboratively modified and distributed buttons, e.g., through email.

2.2 Externalisation

Externalising the design of a piece of software is one of the key quantities that support open authorship. As described in [4, 9], there are two principal components of externalisation.

Firstly, the elements of the design should take an *integral form* — in the language of functional reactive programming, we would say that they give rise to a *signal* or *behaviour*, which has a (in our case complex, structured) value which is in theory observable at any instant in time. Secondly, the elements of the design should have reasonably stable, publicly exposed *coordinates* which may be used to address them. Basman et al. [4] explain that an important property of these coordinates is that they allow for negotiation between different communities, by being structured around references to more or less stable design elements designated as *landmarks*. This is similar to the way in which the structure of CSS selectors in a web document allows for negotiation between code-focused and design-focused authors.

3 CHARACTERISING INTERACTION

Before we investigate the substructure of the phenomenon of interaction, we should try to draw a boundary around the phenomena we are interested in.

3.1 Interaction in the World

There are many theories of interaction with and through computers, some of which are reviewed in [14]. For our purposes, we sketch a broad definition based on Bødker's adaptation of human activity theory for the design of user interfaces [8]. We accept as interaction any situation where a computer is acting as a mediating artefact in some task the human is performing. This mediation can be described at a very high level, such as using computers as tools for writing, as well as lower levels, such as using the mouse-and-cursor assemblage to select and manipulate objects on a screen. From the human user's perspective, software contains many mediators, variously suited or unsuited to the user's goals.

3.2 Interaction as Classically Implemented in Code

For computers and software to become mediators of human action, they need to be able to respond to the outside world. This is broadly the purpose of interactions as implemented in code: To transduce changes at some locations to changes at other locations. When one of these locations is an input or output device, an interaction can involve human users by taking input or producing output and feedback.

However, abstractions for interaction not only determine the form of transduction between locations in a system, but also the means of connecting and disconnecting sources of transduction. We call this aspect *second-order* interaction, since it frames the *first-order* interactions that mediate elementary changes to a system. In this paper, we are interested in how abstractions for interaction enable modification or addition of interactions throughout a system's lifetime. Hence, we focus on second-order interactions.

Listing 1 shows a stereotypical simple interaction as it would be structured in a mainstream imperative language, rendered in a JavaScript-like pseudocode. It exhibits all the typical divergent properties of such code, i.e., it establishes an interaction that is illegible and unchangeable to future users who wish to author their system. For example, the design elements corresponding to the scope and procedure of the interaction (`mouselistener`); the origin and form of the user's operation (`mouseEvent`); and the preconditions for the interaction to begin and end (`interactionStartsCondition` and `interactionStopsCondition`) are all unreferenceable from the point of view of the executing system. Only an author imbued with the god-like powers to fork the original source code and interpose their own definitions can enter this ecology.

More modern, or less imperative, schemes for expressing such interactions, e.g., via functional reactive programming, state machines, or constraint propagation, address some of the infelicities

```
var element = world.findElement(some arguments from design);

var mouseListener = function (mouseEvent) {
    do some stuff with mouseEvent + element
}

if (interactionStartsCondition(some arguments from world) {
    element.addListener('mouseEvent', mouseListener);
} else if (interactionStopsCondition(some arguments from world) {
    element.removeListener('mouseEvent', mouseListener);
}
```

Listing 1: Classical pseudocode operating an interaction

of expression in this example, but do little to expose the resulting system to open authorship without explicit moves towards externalisation.

3.3 The Need to Reconceive Interaction

The above descriptions in turn address the ends (turning computers into mediators of human action), and the means (programming mechanisms such as event listeners) of interaction; but we are equally interested in how a design process draws up an interaction. We believe that the major fault of current approaches to programming interactions is that they do not account for how interactions come to be.

Many decisions have already been made before an interaction can take place: Someone has decided when and where each particular action should be available. For example, when using a word processor of a traditional construction, it is straightforward to edit the presentation of the document being written, but it is usually impossible to edit the presentation of the largely text-based interface around the document. It is also settled how each action can be triggered, whether via a drop-down menu, a toolbar, a keyboard shortcut, etc. Implicitly, interaction is brought into being by software designers and programmers. The consensus "model" of interaction with computers assumes that end users can operate their tools, but that they can only select, organise, combine and share them in delimited ways. On the one hand, the skills necessary to assemble these interactions are typically far away from the application domain in which they are used; on the other hand, even with those skills, it is not generally possible to change those decisions.

Related authorial activities occur when communities bring together various kinds of sub-systems. This integration creates the possibility for encounters between groups of elements which had not been part of the design of either system. It may be the task of a third community to describe and orchestrate such interactions, and each system must be designed in such a way that this third-party integration is possible. This requires suitable public coordinates (as introduced in section 2.2) for each element of both systems that allow the expression of the rules for activating and operating the interaction. Following Kell [15], we propose that user software should form an *integration domain* where useful interactive artefacts can be coupled to construct personal and communal interactive systems.

In the following section, we cut into pieces (anatomise) and examine the process of triggering and enacting interaction, and show how each of the elements of listing 1 can be mapped onto an externalisable design forming a public workflow.

4 ANATOMISING INTERACTION

Meeting our goals of externalisation drives us, at the coarsest level, to anatomise the process of interaction into two distinct activities.

4.1 Interaction as Co-occurrence + Entanglement

First, a specific combination of conditions, which we call a *co-occurrence*, must occur. Co-occurrence determines what elements of the design are in a configuration in which an interaction that involves them may potentially be initiated.

Second, the interaction itself unfolds as an interplay between the elements, which we call *entanglement*. An entanglement is both a process and an object, i.e., a new element that represents the interaction for the duration of its lifetime.

Both phases are necessary to define any interaction, however current systems usually do not distinguish them.

We perform this separation in order to segregate the detection of co-occurrence into its own activity which we expect to be strongly externalisable, since co-occurrence should be a pure function of the coordinates of the existing design elements. This gives rise to a "co-occurrence document" or "co-occurrence signal" which can easily be transported around the system and be worked on by tools of a lower level of sophistication. In the other part of the design we have the entanglements, which will be as externalisable as the implementation technology of the overall system allows — but by being grouped together, offer the potential that they may be treated as first-class elements of the running system, and hence be suitable for participating in further interactions.

The notion of interaction as co-occurrence + entanglement is inspired by the philosophical idea that a tool is not so much a thing in itself as a thing that an agent has brought into a tool-like relation. Interactions are by their nature boundary-crossing: The ability to write is not contained within a pencil nor within a piece of paper, writing is a thing that a person does by tracing a pencil over paper. Hence, we require the ability to detect co-occurrences of things that can engage in a collective behaviour, such as drawing a line.

Separating the description of interaction into co-occurrence and entanglement is necessary in an open ecology of function. The purpose of this separation is to shift away responsibility from programmers, by ensuring that it is no longer their job to fully specify all the entities that will be available in an interactive system throughout its lifetime. Instead of describing interactions in closed terms, that is, with reference to "closed world" quantities such as object handles and event listeners, authors will talk about landmarks in time and space using an open vocabulary in a shared, authorial space. These statements about landmarks will demarcate the creation and destruction of the constituent elements of the interaction, the co-occurrences and entanglements, which themselves can be the subject of future statements.

The following is a blueprint describing the function of these activities in bringing about interaction, and the implementation requirements for interactions to be brought about in a way that conforms with the open authorial principle. We will serve the principle by making sure that we do not need to know all entities that will exist in a given software system at the time it is produced, and furthermore that we do not need to know what interactions they may participate in. This implies that a software substrate for an openly authorable interactive systems can be told to opportunistically recognise the co-occurrence of elements with particular properties, and instantiate entanglements that couple the state of these elements for a time.

In practice, the separation of interaction into co-occurrence and entanglement leads to further separations — we require further elements to represent the activity of detecting co-occurrences, and for instantiating further elements representing entanglements. In the following sections we consider the design of these activities themselves, and how their inputs and outputs are described, generated and consumed. In Figure 1 we illustrate the dataflow in the process as a whole, and label the elements that we argue should be externalised with the numbers 1 to 5.

4.2 State of the (or a) World

Starting at the element numbered 1 in Figure 1, we consider the entanglement workflow to begin with some externalised state, representing the world that a particular community of users are interested in. As we explain in section 2.2, opening up the design process requires this state to have an open structure. Each community of interest will have different concerns that lead them to select representations of the world appropriate to their domains. The direct content of this representation as primary state can be considered as the Model part of the Model-View-Controller triad as initially sketched in [19]. However, for an effectively open system, the selection and representation machinery for this state itself must also be comprised as secondary state within what we consider the "state of the world of interest". This kind of secondary state forms what is termed "device drivers" at the operating system level but has other names depending on the kind of system of interest. Both of these kinds of state should be externalised as openly authorable documents that are available throughout the lifetime of the system, not just at an early design phase.

4.3 Co-occurrence

Before an interaction can begin, there is a prerequisite condition that we call **co-occurrence**: Certain elements of interest must have been assembled in a "suitable proximity" — this proximity may be physical, informational or take some other form that makes the elements conveniently available to each other or to the user.

Here are some examples of co-occurrences that can trigger common interactions:

- Two sensors are plugged into the same machine;
- A finger touches a screen;
- A colour-picking instrument is targeting a particular pixel;
- A user's gaze is being tracked by a camera determining its intersection with an element shown on screen;
- A group of musical instruments is assembled in a sufficiently small area that they may mutually communicate wirelessly;
- A user whose wheelchair carries a device which facilitates communication is brought near a kiosk running an application they wish to interact with.

In each case, the conditions for co-occurrence being satisfied signal that an interaction may potentially begin. The form of co-occurrence is a signal (labelled 3 in Figure 1) containing references to the co-occurring elements, which is emitted while the co-occurrence is ongoing. This signal is derived from the "state of the world" document by an element known as the *co-occurrence engine* (or *function* in the typical case that it acts as a pure function on the document).

4.3.1 *Externalisation of Co-occurrence.*

In order for the co-occurrence to be successfully externalised (section 2.2), it is necessary for these references, and the elements they point to, to be externalisable — this is one of the main drivers leading to the original "state of the world" taking the form of an externalised document, which can be stably referenced using these coordinates.

In a straightforward implementation which has good authorial values, we can then cast the work of the entanglement instantiator as a pure function of the co-occurrence signal, which maps detected co-occurrences into entanglements for as long as they last, and subsequently maps them back into non-existence. This could be described as an "integral" model following the terminology of section 2.2.

This signal can then be conveniently transported around different sites of a distributed system, allowing the instantiation machinery itself to be implemented wherever it is convenient — perhaps at sites where computation is economically available, or perhaps where communication with a crucial system element or the user has particularly low latency. This argument supports the externalisation of the co-occurrence signal itself, labelled 3 in Figure 1.

4.4 Entanglement

After a suitable co-occurrence has been recognised and selected for activation by entering it into the co-occurrence signal, it must be acted upon to initiate an interaction. At this point there are various sets of terminology we will use to describe the process, governed by different sets of metaphors. These are elaborated in section 5, but for the current discussion we will refer to the entity created to represent the interaction process as an **entanglement**. The entanglement has a lifetime coextensive with the interaction, and represents it as an externally addressable element of the system's runtime.

Whilst the beginning of the lifetimes of the co-occurrence and entanglement will invariably agree, there may be interaction models in which the entanglement persists beyond the co-occurrence and follows the agents as they diverge.

One example of such an interaction is manipulating a slider with a mouse cursor, where the initiating co-occurrence is that the cursor and trough overlap, but the drag continues as long as the mouse button is held, even if the cursor moves away from the trough. Note that in this case, the entanglement termination condition could still be best expressed as an externalizable, integral signal, derived from a pure function of the coordinate of the reactants — just a different such signal than the one bracketing the parent co-occurrence.

4.4.1 *Externalisation of Entanglement.*

There are some conditions on entanglements (numbered 5 in Figure 1) that result from our open authorship goals described in sections 2 and 2.1.

Firstly, entanglements themselves must be first-class elements of the system in their own right — and capable of giving rise to further co-occurrences and hence entanglements. As a result, the right-hand side of Figure 1 leads back into the "state of the world".

Externalisation of entanglements allows us to trace what entanglements affect what elements of the world, and vice versa. It also allows entanglements to stretch across all of the externalised

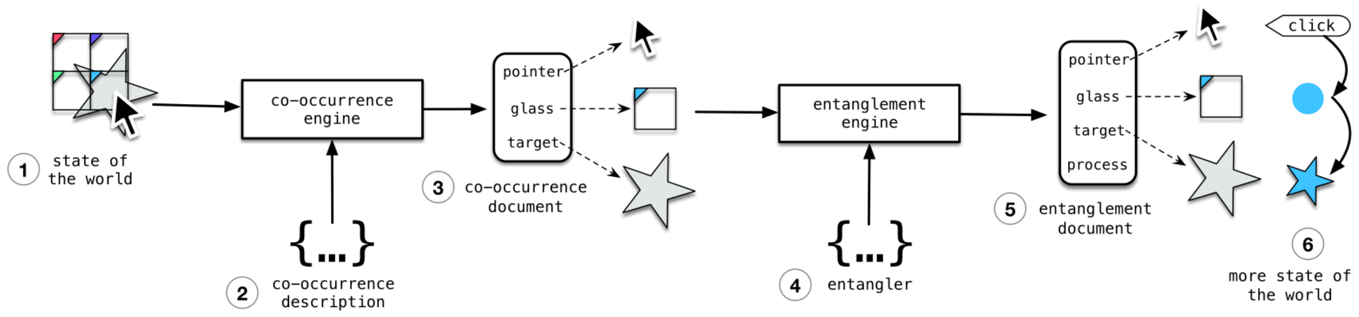


Figure 1: Anatomy of the interaction process, illustrated with Bier et al.'s Toolglass interaction technique [7]. Filled arrows represent dataflow, dotted arrows represent references, and numbered elements are externalised.

world, i.e., they are not bound to only act on the runtime they exist in. Crucially, externalising entanglements supports integrating entanglements created by different authors.

4.5 The Co-occurrence and Entanglement Engines

Our plan caters for a plurality of designs for the co-occurrence and entanglement process, and there are design choices reflecting how much of the signal transformation is performed by the two machines whose descriptions are labelled 2 and 4 in Figure 1.

In the following sections we argue for the externalisation of the descriptions of these engines themselves. We will present arguments for externalisation of similar design elements at the same time, given the somewhat flexible division of their work.

4.5.1 Workflow Connecting Co-occurrence to Entanglement. In any realistically-sized system, such co-occurrences may be detected with high frequency or even continuously. This may require a further round of interaction between the user and the system to detect and transmit the co-occurrences:

The user could select from a menu of available co-occurrences, perhaps with suggestions emphasising choices that they or members of their community of interest have frequently made in the past. Alternatively, a set of rules, e.g., based on priorities, may choose the suitable co-occurrence automatically. In simpler, or classical interactions initiated by means of a dedicated pointing device, it may be appropriate to make the choice of initiating any available interactions immediately.

In all cases, an essential component of the descriptions of the co-occurrence engine and entanglement engine themselves determines what combinations of elements they recognise and how they assemble them into an entanglement complex. These parts of their descriptions could be termed *recipes* for particular entanglements, using a metaphor we will discuss further in section 5.

4.5.2 Supporting Open Authorship with Recipes. We wish to lower the bar to productive use of a community's tools to members of that community as much as possible. Within a well-developed ecology of function, it should be possible to "plug in" known interactions with no programming skills necessary. As a landmark example of what this could look like, *Buttons* [16] allowed users to e-mail each other newly developed tools, which could trivially be arranged in their respective workspaces. *Buttons* buttons have slots for configuring parameters and providing input/targets. When interaction is limited to the button-pressing paradigm, that behaviour can live within the button, but in our more general idiom, behaviour does not have a well-defined scope until one is provided by a co-occurrence. Thus, recipes can engage the elements provided by a particular co-occurrence in a reusable entanglement.

Edwards et al. [12] provides an instructive example of the value of such recipes. In the *Obje* project (formerly known as *SpeakEasy*), Edwards et al. [12] develop an infrastructure to let networked devices be integrated in an ad hoc way by letting them exchange behaviours. For example, the *Obje* infrastructure allowed users to control and connect devices on their local network via a PDA browser. However Edwards et al. did not focus on establishing an *Obje* community, or develop infrastructural features for such a community. Interestingly, they conclude that allowing ad hoc interoperation of devices inherently burdens users with establishing useful semantics for devices they use.

To reduce this interpretive burden, Edwards et al. experimented with a recipe format, called *task-oriented templates*, to combine networked devices offering particular services in a routine configuration [12, p. 3:33]. However, what is a routine configuration depends highly on the particular community of function in which the configuration takes place. Externalising recipes can enable communities of users to collaboratively shoulder the interpretive burden of design, by developing a repository of their useful interactions. This hypothesis is supported by the success of the tailoring community approach of MacLean et al. [16].

5 METAPHORS FOR THE INTERACTION PROCESS

Since this is freshly mapped conceptual territory, several metaphorical structures have been applied to describe the elements operating the interaction process. The base term "co-occurrence" is fairly generic and not tightly bound to any particular mapping, but other metaphors for viewing the nature of the process lead to different names for its elements. These are described in the following subsections and illustrated in table 1.

Description	Chemical metaphor	Quantum metaphor	Cookery metaphor
The characterisation and detection of those elements which might participate in an interaction	Co-occurrence	Co-occurrence	Co-occurrence
The description of the participating elements, the process which they enter into, and the identity of the result		Entangler	Recipe
Machinery which operates the process	Reactor (Co-occurrence engine)	Entanglement engine	
The elements which potentially or actually participate	Reagents	Entanglement components	Ingredients
The process of combining the elements	Reaction	Entanglement	Cooking
The complex resulting from combination	Products	Entanglement	(Dish)

Table 1: Terminology arising from variant metaphors to describe the entanglement process

5.1 Chemical Metaphor

In this metaphorical structure, we describe the prerequisites for the interaction as *reactants* and the resulting structure representing the live reaction as the *product*. This metaphor is helpful in aligning us with the SMIRKS reaction transformation language [11], developed in order to describe not only the molecules which are required for a particular chemical reaction to proceed, but also the exact relationship between atoms in the product and those in the reactants[1]. This is helpful as a generalisation of the regular-expression-like SMARTS language for encoding predicates on molecules, as well as showing an example of how declarative and publicly intelligible notations for reactions can be written and operated by a running system. Areas where this metaphor breaks down for us include that chemical reactants are typically consumed during reactions, and the products are fabricated from their components — whereas for the kinds of reactions we are centrally interested in, the product typically mounts *references* to the reactants, which are typically unchanged by the process of constructing the product itself (although being put into this relation may mediate changes in the reactants during the lifetime of the product).

5.2 Quantum Metaphor

In a metaphor taken from quantum physics, we describe the participants in the reaction complex as being *entangled*, and speak of the resulting complex as an *entanglement*. The machinery that instantiates and transmits descriptions of these complexes is called an *entanglement engine* or *entangler*. This metaphor is helpful in situating the purpose of the complex as being primarily informational, and also one that mediates communications and relationships beyond the timeframe of the co-occurrence: in quantum physics, entangled particles interact even after they have been separated, until an external event breaks the entanglement. Similarly, two objects may continue to interact after the co-occurrence has ceased to exist, as when continuing to drag a slider even when the cursor moves away from the trough. This metaphor better maps the "reversible construction" aspect of the entanglement complex than the chemical metaphor, but provides fewer implementation clues.

5.3 Cookery Metaphor

In this metaphor, the available elements are considered as ingredients in a recipe. The co-occurrence of particular groups of ingredients may make the cooking of different dishes available. This metaphor is more useful at the level of the user of an overall system, who may well be familiar with the process of rummaging through their kitchen cupboards in order to determine what ingredients have co-occurred there. It may be that the user habitually wishes to cook particular dishes when faced with a particular co-occurrence, or instead wishes the system to surprise them through suggesting a previously unvisited combination. This metaphor suffers from the same deficiency as the chemical one, in that the ingredients are considered to be irreversibly consumed through the reaction, which is an unsuitable disposition in an open informational ecology.

6 EXAMPLES OF ENTANGLEMENT AND CO-OCCURRENCE ENGINES

We have built several examples of engines operating interactions, which highlight different aspects of our ultimately desirable idiom with varying degrees of fidelity.

6.1 The Co-occurrence Engine

This implementation[2], simply named "the co-occurrence engine" since at the time it was the only such implementation, is aimed at facilitating the use of multiple physical devices and sensors coordinated at or near the same device. As such it currently operates a relatively coarse-grained co-occurrence criterion based on the simultaneous presence of elements advertising particular capabilities, encoded as namespaced strings.

The goal of the co-occurrence engine is to enable dynamic configuration within an Infusion [13] system, based on the presence or absence of Infusion components. We have used it with the GPII Nexus [10] to facilitate exploration of designs for an inclusive science lab (see section 7).

The Nexus 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 provides a means for developers to externalise their application or component's model, enabling it to be observed and modified by other components. The Nexus is being developed as part of the Prosperity4All Project [18], a European Commission-funded project that aims to reduce the cost and complexity of building assistive technologies and adaptive user interfaces.

The co-occurrence engine monitors a collection of Infusion components and is configured with a set of recipes. Each recipe contains

[1] An early precedent for the idea of using chemical reactions as a computational model appeared in [6]

[2] Documentation for the co-occurrence engine can be read at https://github.com/simonbates/co-occurrence-engine/blob/master/documentation/CoOccurrenceEngine.md

```
{
    "type": "gpii.nexus.recipe",
    "reactants": {
        "phSensor": {
            "match": {
                "type": "gradeMatcher",
                "gradeName": "gpii.nexus.atlasScientificDriver.phSensor"
            }
        },
        "collector": {
            "match": {
                "type": "gradeMatcher",
                "gradeName": "gpii.nexus.scienceLab.collector"
            }
        }
    },
    "product": {
        "path": "sendPhSensor",
        "options": {
            "type": "gpii.nexus.scienceLab.sendPhSensor"
        }
    }
}
```

Listing 2: A Co-Occurrence Engine Recipe from the Nexus Science Lab

two sections: a set of reactants and instructions for constructing a product. Each reactant has a name and a match rule. The reactant name may be used within the product to refer to the matched component. In the current implementation, a single match rule is available: match components based on namespaced strings (called "grade names") present on the components. An example recipe from the Nexus Science Lab is shown in Listing 2.

The co-occurrence engine constructs and maintains product components based on the co-occurrence of reactants. If components are added that bring into being new co-occurrence situations with matching recipes, the corresponding products are constructed. If components are destroyed that remove previously present co-occurrences, the affected products are also destroyed — it is thus an implementation of the integral co-occurrence model described in section 4.3.1.

6.2 The Entanglement Engine

This implementation is intended to demonstrate the quantum metaphor as a general-purpose substrate for user software. It focuses on representing interaction techniques in a form that can can be dynamically integrated into a running system. The Entanglement Engine mimics the role of the CSS engine in a web browser, transposed from managing the presentation of content to managing its behaviours.

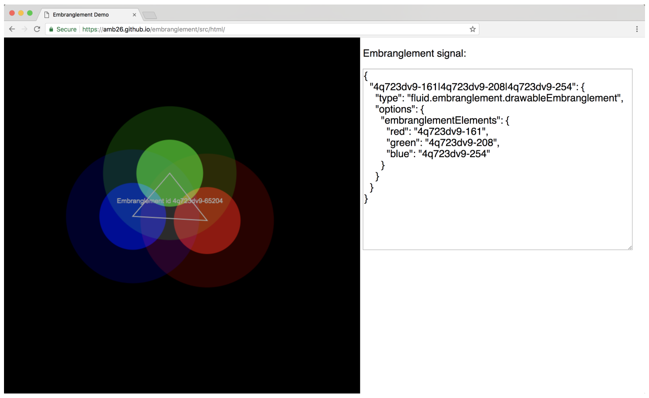
Rather than operate on web documents, the Entanglement Engine is implemented to operate on a structured data world inspired by the Document Object Model (DOM), extended to also represent browser-external elements, such as mouse cursors and input devices. The engine is configured with *entanglers*, which are similar to Co-Occurrence Engine recipes, with a few notable exceptions:

Firstly, if an entangler is triggered, the engine produces an entanglement, a relationship coupling the state of the entangled elements, which are called components. These relationships currently take the form of programming abstractions appropriate to the concrete interaction, e.g., event listeners, reactive functions, state machines [1], etc.

Secondly, entanglers include a list of predicate functions that must hold before an entanglement is created. These predicates are used to flexibly define different kinds of co-occurrence. For example, one entangler may require that its components geometrically overlap, while another may require that one component contains a reference to another.

The set of available co-occurence signals is currently small, but our intent is that entanglers can be configured to trigger based on the signals appropriate to various interaction paradigms, e.g., geometric overlap for conventional mouse pointer-based interaction, device proximity for cross-device systems, or gesture detection for gestural interaction.

Listing 3 shows two entanglers. `mousemove` entangles a mouse and another element which should contain a two-dimensional position. The entanglement it creates adds the relative movements produced by the mouse to the position of the moved component. If triggered opportunistically, this entangler would connect all mice to all position elements. Instead, it is operated by the second entangler in listing 3, `makecursor`. `makecursor` triggers when a mouse exists that does not already have a `mousemove` entanglement. As mice are added to the system, this entangler creates a new cursor for each mouse and instantiates a new `mousemove` entanglement with them as components. Other entanglers allow the created cursors to, e.g., target and move around elements they overlap.


Embranglement Demo
Secure
https://amb26.github.io/embranglement/src/html/
Embranglement id 4q723dv9-65204

Embranglement signal:

```
{
 "4q723dv9-161|4q723dv9-208|4q723dv9-254": {
  "type": "fluid.embranglement.drawableEmbranglement",
  "options": {
   "embranglementElements": {
    "red": "4q723dv9-161",
    "green": "4q723dv9-208",
    "blue": "4q723dv9-254"
   }
  }
 }
}
```

Figure 2: Screenshot of in-browser embranglement engine at the point where an embranglement has been instantiated

6.3 The Embranglement Engine

This is a proof-of concept implementation (using the term "embranglement" in place of "entanglement") which operates in a toy world of three agents in a web browser[3]. It was designed to clearly exhibit the signalised form of the co-occurrence condition, which is displayed as a JSON document in the right pane of the UI, updated at each frame of the interaction. Figure 2 shows a screenshot where the three agents have been deemed to co-occur and an embranglement has been instantiated coupling them together. In this case the co-occurrence condition is geared through "focus/nimbus" relations. Benford and Fahlén [5] describe a model of interaction where an agent has multiple zones of perception — the "focus" of an agent is the region of its own perception, and its "nimbus" is the region from which it can be perceived.

This illustration shows three point agents, coloured red, green and blue which each have a focus and nimbus which are circular and concentric, with the focus nested within the nimbus. The agents enter a 3-way entanglement when the focus of each intersects the nimbuses of the other two. The construction state of the

[3]The engine may be viewed live at https://amb26.github.io/embranglement/src/html

entanglement is illustrated by a white triangle joining the agents, with the unique identifier of this particular entanglement instance annotating it. This implementation was made in the Infusion system [13] developed by the Fluid project, a configuration dialect which makes substantial use of JSON structures and the natural alignment and coordinate system they induce in order to support the aims of open authorship. As a result, the entanglement is indeed on an even footing with the original agents, and meets the criteria in section 4.4 for being the basis of further co-occurrences, as well as being signalised in a form that could easily be advertised and manipulated outside the browser process hosting it.

In this model of interaction, we can simplify the transmission of the state of co-occurrence and entanglement, since this state is simply a function of the position and orientation of all the agents. This state can then be easily transmitted through a distributed system in the form of the JSON document ("embranglement signal") shown in the right pane. As we note in section 4.3, not all interactions may

```
// An entangler for wiring up something with a position to move with a mouse.
export const mousemove = {
    name: 'mousemove',
    // we will entangle two elements:
    // 'mouse', which must have the mouse type,
    // and 'moved', which must have an attribute named 'position'
    // that has the type '2d-coordinate'
    components: {
        mouse : ':mouse',
        moved : '[position:2d-coordinate]'
    },
    on_start: 'reaction getHash -> moveCursor',
    actions: {
        getHash: function() {
            return hashCode(this.find('mouse'));
        },
        moveCursor: function(hash) {
            // function body omitted
        }
    },
}

// An entangler that spawns a cursor for each mouse that connects to the system.
export const makecursor = {
    name: 'makecursor',
    // we will entangle one element, 'mouse', which must have the type 'mouse'
    components: {
        mouse: ':mouse'
    },
    // the 'mouse' element should not already be entangled with a cursor
    configuration: ['mouse hasNoCursor'],
    on_start: 'makeCursor',
    actions: {
        hasNoCursor: function(components) {
            // function body omitted
        },
        makeCursor: function(components) {
            // create a cursor
            let cursor =
                d(':circle.mousemove-moved.shapefocus-focus.dragging-leader', {
                fill       : d(':color', 'transparent'),
                stroke     : d(':color', '#000000'),
                strokewidth: d(':number', 2),
                radius     : d(':number', 8),
                position   : d(':2d-coordinate', {x: 0, y: 0}),
                targets    : d(':id-list', [])
            });
            // add it to the document
            let cursorMountPoint =
                findFromNode(components.mouse, ':root', ':renderables')[0];
            cursorMountPoint.get('content').push(cursor);
            // entangle the cursor and the mouse using the mousemove entangler
            this.entangle({
                mouse: components.mouse,
                moved: cursor
            }, mousemove);
        }
    }
}
```

Listing 3: Two Entanglement Engine entanglers

be signalised straightforwardly in this way, although an Infusion component tree as a whole enjoys a natural externalisation which could still be relied on in the case of interactions that lack a natural signal form.

7 CONCRETE APPLICATION EXAMPLE

Here we will describe how one of our engines, the co-occurrence engine of section 6.1, has been employed to construct a system and interaction meaningful to end users in a physical context, the Nexus Science Demo[4]. The demo implementation is not at a stage where it substantially meets the goals of open authorship described in section 2.1, but we describe how our factoring of the interaction process leads to an open, flexible design supporting variant presentations of data collected from dynamic sources and how we plan to extend the design towards more open authoring of interactions.

7.1 The Nexus Science Demo

The goal of the Nexus Science Demo is to explore designs that make science labs more inclusive. We use the GPII Nexus [10] to connect lab sensors to a range of different presentations and interactions. This helps a student pick the interaction that works best for them. For some students this might be a visualisation and for others it might be a sonification, or a combination of different presentations.

Drivers for three sensors were developed: a USB pH sensor, a USB electrical conductivity sensor, and a Raspberry Pi based temperature sensor. Six presentations were developed: a numeric dashboard showing the values of all connected sensors, a bar-graph visualisation, a coloured pH visualisation annotated with the values of common substances, a coloured temperature visualisation, a general ranged-value sonification, and a pH sonification.

The sensor drivers work by maintaining peers within the Nexus. When a sensor becomes available (for example by connected a USB based sensor to a computer running a suitable driver, or by adding a Raspberry Pi to the network), a peer is constructed within the Nexus, and a WebSockets connection is established to stream sensor value updates. When a sensor becomes unavailable, the peer is destroyed. In this way, the availability of each sensor is indicated by the presence or absence of a peer within the Nexus.

7.2 Co-occurrence for Open Designs

The needs of open authorship are a superset of those which lead to open, configurable application designs. We describe how the current science demo supports flexible aggregation of an open collection of live data sources, and then how we plan to extend such designs into systems supporting authorship of such demos supporting the needs of individual learners.

The co-occurrence engine is used to decouple the sensor drivers from the presentations that students use to interact with the data. The sensor drivers need only make their data available and the co-occurrence engine recipes organise the data for the presentations. Presentations expect a central data store with an entry for each sensor, together with metadata such as name and value range. The co-occurrence engine is configured with one recipe per sensor type, each with 2 reactants: the sensor that the recipe is for, and the

[4]A video of the demo in action can be found at https://www.youtube.com/watch?v=NNwc0VYRhUU

presentation collection data store. When these reactants are present, the recipe constructs a product that relays the sensor data to the collection data store, in a format expected by the presentations.

The co-occurrence engine facilitates dynamic connection and disconnection of sensors, as student needs change. When a new sensor is connected, the driver will construct a peer for the sensor. The co-occurrence engine will then construct any product components needed to make the sensor data available for presentation. Once the sensor data is available, the student may then select from the available data presentations. When they are finished, they may disconnect the sensor. The driver will destroy the peer and the co-occurrence engine will tear down the products that it had previously set up.

7.3 Extending the Design

The goal of the design work underlying the Science Demo, to feed into the education activities in the FLOE project[5], is to support one-to-one learner customisation. We will build tools not only to allow content authors to author multimodal content for consumption by learners, learners themselves will make custom visualisations and sonifications, presentations of data that are meaningful to them, that they can share and remix with other users. This will form a substrate that we term the "Materialisation Toolkit".

8 CONCLUSION

We have introduced a new taxonomy for the constituents of interaction, and set new goals that must be met by systems offering it, in order to support open ecologies of use and construction through open authorship. We have argued that our reconception of the process of interaction can support more externalised designs, and a more open set of choices for who can make decisions about how a system should be, and how it is mapped onto the world. We have exhibited variant metaphors and variant concrete implementations for such systems, contrasting the relevance of each for different tasks and contexts.

Our goal has been to develop an anatomy of interactions that conforms to the open authorial principle. If interactive software is to truly support open ecologies of function and fabrication, interactions should be constructed as additions to an already-existing system. Our implementations are currently at various stages from proof of concept to early prototype. The true test of these implementations is going to be integrating them into communities of practice. We will continue to refine this proposed anatomy and pursue the elusive goals of open authorship.

ACKNOWLEDGMENTS

This work was partially supported by European Research Council (ERC) grant n° 695464 ONE: Unified Principles of Interaction.

REFERENCES

[1] Caroline Appert and Michel Beaudouin-Lafon. 2006. SwingStates: Adding State Machines to the Swing Toolkit. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*, Jeff Pierce (Ed.). ACM, New York, NY, USA, Article 1. https://doi.org/10.1145/1166253.2180954

[2] Antranig Basman. 2016. Building Software is Not a Craft. In *Proceedings of the Psychology of Programming Interest Group.*

[3] Antranig Basman, Luke Church, Clemens Klokmose, and Colin 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] Antranig Basman, Clayton Lewis, and Colin Clark. 2018. The Open Authorial Principle: Supporting Networks of Authors in Creating Externalizable Designs. In *Submitted to Onward '18.* https://github.com/amb26/papers/blob/master/onward-2016/onward-2016.pdf

[5] Steve Benford and Lennart Fahlén. 1993. A Spatial Model of Interaction in Large Virtual Environments. In *Proceedings of the Third Conference on European Conference on Computer-Supported Cooperative Work (ECSCW'93).* Kluwer Academic Publishers, Norwell, MA, USA, 109–124. http://dl.acm.org/citation.cfm?id=1241934.1241942

[6] Gérard Berry and Gérard Boudol. 1992. The chemical abstract machine. *Theoretical computer science* 96, 1 (1992), 217–248.

[7] Eric A. Bier, Maureen C. Stone, Ken Pier, Ken Fishkin, Thomas Baudel, Matt Conway, William Buxton, and Tony DeRose. 1994. Toolglass and Magic Lenses: The See-through Interface. In *Conference Companion on Human Factors in Computing Systems (CHI '94).* ACM, New York, NY, USA, 445–446. https://doi.org/10.1145/259963.260447

[8] Susanne Bødker. 1991. *Through the Interface - a Human Activity Approach to User Interface Design.* Vol. 16. Lawrence Erlbaum Associates, Hillsdale, NJ.

[9] Colin Clark and Antranig Basman. 2017. Tracing a Paradigm for Externalization: Avatars and the GPII Nexus. In *Companion to the First International Conference on the Art, Science and Engineering of Programming (Programming '17).* ACM, New York, NY, USA, Article 31, 5 pages. https://doi.org/10.1145/3079368.3079410

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

[11] Inc. Daylight Chemical Information Systems. 2008. SMIRKS - A Reaction Transform Language. (2008). http://www.daylight.com/dayhtml/doc/theory/theory.smirks.html

[12] W. Keith Edwards, Mark W. Newman, Jana Z. Sedivy, and Trevor F. Smith. 2009. Experiences with Recombinant Computing: Exploring Ad Hoc Interoperability in Evolving Digital Networks. *ACM Trans. Comput.-Hum. Interact.* 16, 1, Article 3 (April 2009), 44 pages. https://doi.org/10.1145/1502800.1502803

[13] Fluid. 2018. Fluid Infusion Documentation. (2018). http://docs.fluidproject.org/infusion/development/

[14] Kasper Hornbæk and Antti Oulasvirta. 2017. What Is Interaction?. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17).* ACM, New York, NY, USA, 5040–5052. https://doi.org/10.1145/3025453.3025765

[15] Stephen Kell. 2009. The Mythical Matched Modules: Overcoming the Tyranny of Inflexible Software Construction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09).* ACM, New York, NY, USA, 881–888. https://doi.org/10.1145/1639950.1640051

[16] Allan MacLean, Kathleen Carter, Lennart Lövstrand, and Thomas Moran. 1990. User-tailorable Systems: Pressing the Issues with Buttons. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90).* ACM, New York, NY, USA, 175–182. https://doi.org/10.1145/97243.97271

[17] Fabio Paternò and Volker Wulf. 2017. *New Perspectives in End-User Development.* Springer.

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

[19] Trygve Reenskaug. 1979. THING-MODEL-VIEW-EDITOR - an Example from a planningsystem. Technical note, Xerox PARC. (May 1979). http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pd

[20] Peter Wegner. 1997. Why interaction is more powerful than algorithms. *Commun. ACM* 40, 5 (1997), 80–91. https://doi.org/10.1145/253769.253801

[5] https://floeproject.org/