Critique of ‘An Anatomy of Interaction: Co-occurrences and Entanglements’

Tomas Petricek
University of Kent
Canterbury, UK
tomas@tomasp.net

ABSTRACT

The paper by Basman et al. suggests that we think about programming in terms of *interaction* rather than *algorithms*. This call needs to be interpreted in a broad sense – the idea of interaction is not just another programming abstraction, but different way of structuring our thinking about programming. This includes thinking about how users can interact with the software more generally, but also what are effective metaphorical ways of thinking about software.

In this critique, we review some of the core ideas presented by Basman et al. We consider what programming *substrate* might be used to implement the systems proposed by Basman et al. That is, systems that blur the boundaries between users and developers. We also review a number of systems that are technically similar to co-occurrences and entanglements and we reconsider them through the perspective of the research paradigm based on interaction.

CCS CONCEPTS

• **Software and its engineering → General programming languages**; • **Human-centered computing → Interaction design theory, concepts and paradigms**;

KEYWORDS

Programming paradigms, interaction, end-user programming

ACM Reference Format:
Tomas Petricek. 2018. Critique of ‘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, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Anatomy of Interaction by Basman, Tchernavskij, Bates and Beaudoin-Lafon [1] considers programming of computer systems in terms of *interaction*. At the technical level, concepts such as *events* appear in a number of programming languages and libraries. What makes the work by Basman et al. interesting is that it is not focused solely on the technology, but considers *interaction* as a foundational concept.

In computer science, the idea of an *algorihm* became the foundation of a dominant research paradigm in 1960s. Ensmenger [6] argues that this framing of computer science served the community well within the university, as it provided a academically respectable mathematical method for talking about computing. It also determined what problems are studied and what questions can be asked.

The work by Basman et al. can be seen as an attempt to reframe the foundations of programming research and rebuilding it on the concept of *interaction*, rather than on the concept of *algorithm*. This has the potential to change the technical aspects of how we write software, but it also lets us see software from a wider socio-technological perspective. We can ask not just “how can interactions be implemented”, but also “who can interact with a system and how” or “how are interactions conceptualised by the actors”.

This critique revisits some of the ideas mentioned in this paper and related work through the perspective of this new imagined research paradigm.

- In Section 2, we consider the aim of enabling *active participation* where users can freely modify a system. This requires providing a simple way of specifying interaction logic. How might this look and how do we assess simplicity?
- Basman et al. refer to systems that can be modified within themselves as *open systems*. In Section 3, we speculate on fundamental limitations of such systems.
- In Section 4, we review three metaphors for thinking about interaction proposed by Basman et al. and argue for the importance of metaphors in programming research.
- Finally, in Section 5, we look at work that proposes concepts similar to co-ocurrences and entanglements. We look at systems that are related at the technical level. We consider them through the perspective of the research paradigm based on interaction.

2 ENABLING ACTIVE PARTICIPATION

The authors note that there is currently a large gap between users and creators of software. Even if the software is open-source, making even a simple change such as correcting typos is inaccessible to most users. They argue for an *open ecology of function* – the system should enable users to modify and extend it. In this critique, we consider two aspects of the idea – first, what is the *simplicity* that would enable such openness and, second, what are the *substrates* through which it can be provided.

2.1 Understanding Simplicity

Much of programming language research aims to make programming *simpler*, but we do not have a very good understanding what this means. If we want to make simplicity – which is essential for the open ecology of function – a legitimate academic topic, we need to find a way of talking about it. This topic is not discussed by Basman et al., perhaps because when you see a simple solution, you know that it is simple! Perhaps, but this is too subjective claim.

Author version
<Programming’18> Companion, April 9–12, 2018, Nice, France
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$0.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

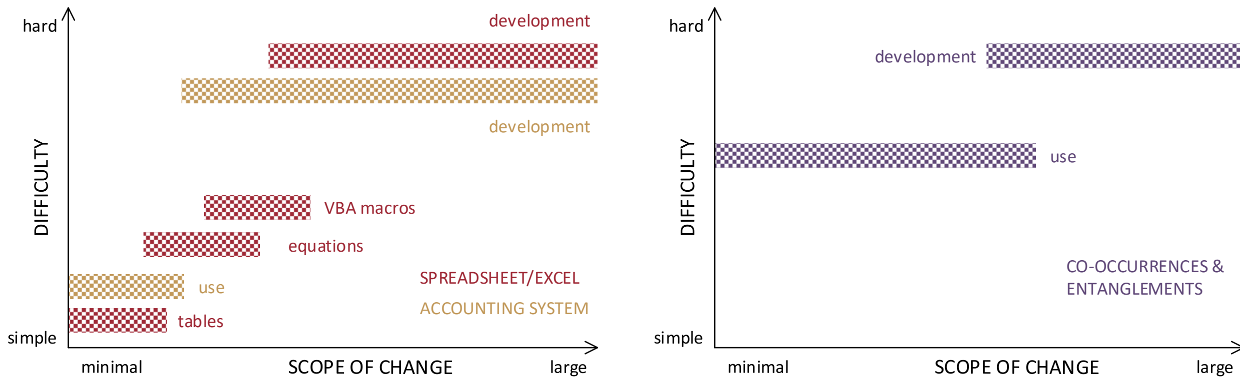

Figure 1: An illustration of different substrates for modifying systems. The diagrams show the difficulty of making changes to a system using different substrates. On the left, we compare typical application (with use and development) with a spreadsheet system that provides multiple substrates. The dagram on the right shows a worst-case scenario for a co-occurrences & entanglements system. The diagrams are merely illustrative and the scales are informal. Each substrate has one fixed difficulty and covers a range of changes it supports.

How can we recognise and talk about simplicity? First, it is possible that simplicity is *tacit knowledge*, i.e. a kind of knowledge that is gained through experience and personal involvement with the subject matter. As described by Polanyi [12], this kind of knowledge cannot be explicitly written down and analysed. Polanyi gives an interesting example of the common and statory law systems. Statutory law system is specified as a set of written rules that judges follow. The common law system provides a set of cases with past decisions, rather than written rules. The idea is that the rules are *tacit knowledge* that cannot be fully written down, but can be understood through the cases. If we see simplicity in programming as tacit knowledge, we would still be able to judge it through a set of cases, or case studies, that demonstrate a number of (more or less) open systems, describe changes made by their users and judge their simplicity.

Second, it might be the case that our current research paradigm, based on mathematical and scientific analysis on programs, does not equip us with good tools for understanding simplicity, but an alternative paradigm would. One such paradigm is aluded to in the original paper – if we analysed programming concepts through the perspective of cognitive science, we could argue that the simplicity of a programming model depends on the metaphors it requires. As such, programming model that can be explained in terms of cooking is easier to understand than a model based on a machine that writes symbols on an infinite tape.

2.2 Editing Substrates

The idea of *open ecology of function* aims to erase the gap between the developer of a system (who has full control) and the user of the system (who is limited to operations that the developer imagined). The gap exists because both personas operate using two different *substrates*. Developers interact with the *source code* of the system while users interact with the *user interface* of the system (which might be graphical or text based).

The division between the user and the developer is common and clear, but there are systems that provide multiple *substrates* that may be accessible to multiple personas. For illustration, see Figure 1 (left). The diagram compares a typical application (e.g. for accounting) with a spreadsheet system such as Excel. Spreadsheet systems provide multiple substrates – user interface for entering data, language for specifying formulas, scripting language for creating macros (such as VBA) and the source code of the system itself. What does a spreadsheet system teach us about providing an *open ecology of function*?

- Support for macros and scripting makes the system more open. We can make larger changes to the system than a regular application enables.
- Macros are not an end-user programming tool, but require expert knowledge. (Although research on end-user programming might make some macros accessible to non-programmers.)
- Macros are not easy to share and re-combine. Copying a specific functionality from one spreadsheet with multiple macros to another is a difficult programming problem.
- Spreadsheets mix multiple substrates. The difference between formula, macro and system langauge is perhaps incidental, but there is a notable difference between entering data and writing code.

A system based on co-occurrences and entanglements faces a danger of providing the structure of substrates illustrated in Figure 1 (right). The aim of providing substrate that allows large change during regular usage of the system might make the usage of the system more difficult. At the same time, if the system is not built "using itself", there will still be changes that require modifying its source code in a traditional way. The lessons for building systems that provide an open ecology of function are:

- The system should be built using just one substrate that allows small changes, as well as large changes. We return to this topic in Section 3.

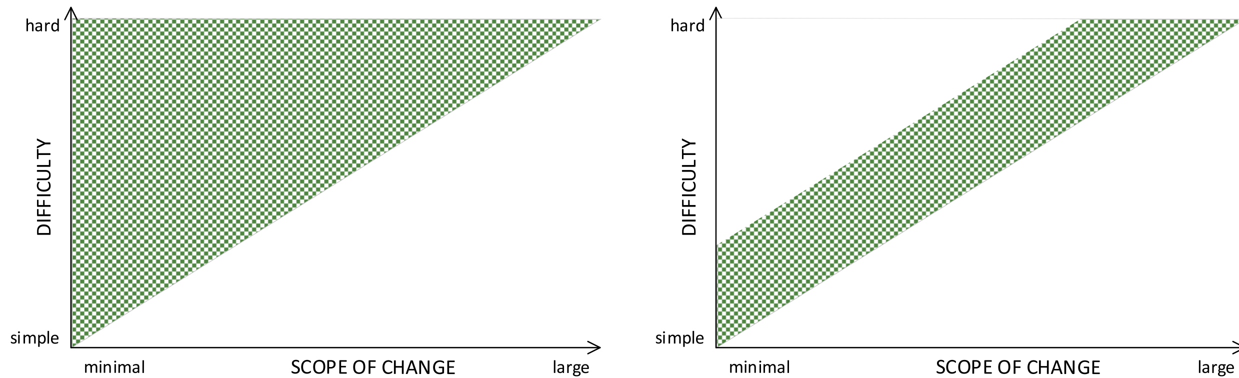

Figure 2: Structure of substrates that enable open systems. An open system based on a single substrate must make simple changes easy while still allow large changes. Large changes will never be easy, which is why the area below a diagonal is always blank. The substrate on the left allows all kinds of changes, but even a simple change may be difficult to do (if the substrate is not used correctly). Can substrate restrict the complexity of simple changes (on the right) to make those always easy?

- How to make small changes simple remains an open question. However, it is important that the single substrate can be edited in multiple ways – through a (structured or text-based) interface that is suitable for more complex changes and through a (graphical) interface suitable for simple changes. However, the two interfaces must use the same substrate.
- The architecture needs to make the individual pieces of functionality easily shareable and re-combinable. Using *co-occurrences* and *entanglements* makes this easier as long as they can be copied between systems without having to change the representation of the state of the world.

3 DESIGNING OPEN SYSTEMS

The proposed system should not just enable limited modification (such as macros in Excel), but it should support the *open authorial principle*, i.e. *"an expression by one author can have its effect replaced by another author"*. This can only be achieved if the system is "created within itself" – otherwise, the substrate will always place constraints that cannot be escaped. If a system provides a macro mechanism, it is likely impossible to replace the macro mechanism by writing a new macro.

3.1 Limits of Open System Complexity

A system written in itself is a theoretically appealing idea, but we hypothesize that it might pose a too strict and unnecessary requirement. We might prefer a system supporting open ecology of function that allows authors to replace *almost* all other effects, over a system that is too complex. Yet, it is possible to create such fully open systems. In Smalltalk, the development environment is fully constructed within itself and it allows authors to replace and modify all of the existing functionality.

Smalltalk [8], however, is an expert programming environment that does not make small changes simple to non-developers. This might be a principal limitation of fully open systems. A system that can be implemented in itself needs to provide a certain minimal expressive power, but this, in turn, means that it will have a certain minimal complexity. (Much like Turing completeness leads to undecidability in the research paradigm based on algorithms.) We find understanding such limits of complexity open systems an important research problem for the proposed programming paradigm based in co-occurrences and entanglements.

3.2 Making Small Changes Simple

An open system needs to be constructed using a substrate that allows both large changes (those will inevitably be difficult to make) and simple changes (these should be simple to make). Figure 2 (left) shows a diagram (similar to those in Section 2.2) that illustrates the scope that a substrate of open systems must cover. As discussed before, to allow simple small changes as well as complex difficult changes, the user should be able to interact with the substrate in multiple ways, possibly including a graphical interface and textual interface.

The diagram on the left also highlights one danger of such generic substrates. In particular, even a small change can be very difficult if it is not done in a suitable way. It seems difficult to design system that allows complexity (to enable large changes) but prevents using the same complex methods when implementing a small change. Ideally, the substrate diagram would look as in Figure 2 (right), where the substrate is close to the diagonal – simple changes are easy, large changes are difficult (but possible), but small changes cannot be done in a complex way.

4 PROGRAMMING AND METAPHORS

The authors suggest that the idea of co-occurrences and entanglements has been explained in terms of three metaphors in past work. The cooking metaphor treats program as a recipe, chemical

```
search
  students = [#student]
  total-students = count[given: students]

bind @browser
  [#div text: "{{total-students}}
    are in the school district"]
```

Figure 3: Counting students with Eve. Eve programs are written as blocks consisting of search clause and bind clauses. Search specifies requirements for triggering the pattern (akin to co-occurrence) and bind specifies actions that are produced (akin to entanglement). In this example, search collects all students and counts their total number while bind updates information on a web page.

```
def get(c) & putInt(n) =
  c(sprintf "Number: %d" n)

def get(c) & putString(s) =
  c(sprintf "String: %s" s)
```

Figure 4: One-place buffer in JoCaml. In JoCaml, programs are specified as *joins* between one or more *channels*. A pattern is triggered once calls to all joined channels are made (thus a join pattern is akin to co-occurrence). Here, the `putInt` and `putString` channels provide two ways of putting values into a buffer and `get` takes a continuation that will be triggered with a composed message once a string or an integer is put into one of the two put channels (adapted from [11]).

metaphor treats interaction as a chemical reaction and quantum metaphor explains how objects remain interlinked as a result of entanglement.

We find the discussion about metaphors noteworthy. Lakoff and Núñez [9] convincingly argue that metaphors are the key for construction of (even abstract) mathematical thought and we similarly find metaphors important for thinking about programming. Petricek [10] argues that each programming concept should be understood at three levels – *formal* used for reasoning, *implementation* representing source code and *metaphorical* providing the intuitive understanding. In this light, discussion about metaphors should be an inherent part of any programming paper.

4.1 Metaphors for Interaction

The idea of *interaction* is more directly rooted in everyday experience than the idea of *algorithm*, which might make it more suitable for metaphoric thinking.

The use of multiple metaphors for describing the one family of systems is akin to some of the metaphors used to construct mathematics in the work of Lakoff and Núñez. For example, numbers and arithmetic can be seen both as an *object collection* and as *movement along a path*. Using multiple metaphors makes it possible to understand cases where one metaphor does not work well. For example, if we interpret numbers as collections of given sizes, it is difficult to interpret what zero means (there is no *empty collection* in the real world), but if we see numbers as movement, then zero corresponds to staying in the inital place.

In the same way, multiple metaphors for co-occurrence and entanglement might provide complementary ways of looking at the system. Perhaps most importantly, they might give us a more "user" and a more "expert" perspectives on the same substrate. We suggest that a hypothetical system might use a simple metaphor such as cooking to allow users to make simple changes and a more complex metaphor such as quantum physics to allow experts reshape the system more significantly. Such combination might allow us to design a system that is close to the one imagined in Figure 2 (right).

4.2 Methodological Questions

The study of metaphors in programming is relatively new and much remains to be done. Metaphors are often used when explaining abstract programming concepts. Some argue that this is a mere kludge, while some see such metaphors as fundamental for our understanding. This question is likely a "wicked problem" [13] that cannot be easily empirically tested. One aspect of metaphors that has been empirically tested is whether they help understanding in certain narrow contexts such as visual programming [3] or the use of diagrams [2].

The paradigm envisioned by Basman et al. has the potential to make metaphors an important programming design tool, but it also shows we need better theoretical tools for analysing metaphors in the context of programming:

- First, we need to study how metaphors relate programs to the real world experience. Lakoff and Núñez talk about *grounding metaphors* that link physical experience with basic mathematical concept (sets are like physical containers) and *linking metaphors* that relate multiple mathematical entities. Are those concepts applicable to programming, or do we use different kinds of metaphors?
- Just like arithmetics as *object collection* does not naturally explain zero, the three metaphors mentioned by Basman et al. each have some limitation in how well they explain co-occurrence and entanglement systems. What is a metaphor that explains the system the best? And should we compare metaphors in this way?

5 RELATED WORK

The Anatomy of Interaction can be seen from multiple perspectives. In Section 2 and Section 3, we discussed the human centric perspective and the aim to enable wider participation. In Section 4, we focused on cognitive aspects of the technology of co-occurrences and entanglements. In this section, we wrap-up by considering the technical perspective.

The idea of a co-occurrence is similar to programming abstractions from a number of reactive and concurrent programming systems. Although those are mostly aimed at expert developers, it is worth pointing out the technical similarities.

5.1 Joins and Pattern Matching

To quote the authors, *"co-occurrence determines what elements of the design are in a configuration in which an interaction that involves*

them may potentially be initiated”. Aa number of past systems implement a limited form of co-occurrence detection. Pattern matching in functional programming also checks for a specific configuration, but considers only a shape of given input data structures.

Join calculus. Functional patterns have been generalised to *join patterns* [4] in programming languages based on the join calculus such as $C\omega$ or JoCaml [5]. Join patterns are closer to co-occurrences because they also include temporal aspect – a join pattern is detected each time a specific configuration of values in channels is available. For example, see Figure 4, which shows a one-place buffer implemented in JoCaml. Patterns such as `get(c) & putInt(n)` can be seen as co-occurrences, with `c` and `n` being variables that can be used to construct an entanglement.

Although join calculus appeared as a theoretical abstract machine, it is worth noting that it shares the chemical metaphor [7] suggested by Basman et al. Therefore, join calculus is related work both at the implementation and at the metaphorical level.

Logic programming with Eve. Eve [14] is a system shares the aim of allowing more people to program, although it does not necessarily share the aim of making all systems open to modification by users. Technically, it is based on logic programming. Its core is similar to pattern matching – a code block can specify `search` clause which declaratively specifies a required configuration. The `bind` block then specifies what to do in response. An example is shown in Figure 3. Interestingly, Eve documentation does not describe the architecture in terms of metaphors, but it largely matches the architecture of co-occurrence and entanglement systems and might provide another fruitful example.

5.2 Opening Closed Systems

All of the aforementioned programming concepts were introduced as programming languages or language features for software developers. They were not intended as the foundation of a system that provides an *open ecology of function.* However, is it possible to take such closed system and make evolve it into an open one?

At the technical level, the system needs to allow modification of the source code by the user (during execution). This seems feasible for systems such as JoCaml or Eve. The architecture of pattern matching (or co-occurrences) that operate on some global state make such dynamic code update easier as we do not need to interrupt executing program.

We still need to make editing of code easier in order to make the system fully open to user modification. As discussed before, this might require us to provide multiple ways of editing the same substrate – and both join patterns and logic programming as in Eve might provide a suitable substrate (expressive enough to satisfy the expressive limit described in Section 3.1).

Taking an existing system and making it open might also provide a way to bootstrap a community around an open system. Rather than building the whole system and community around it from scratch, we might prefer to build on existing tools and community interest to turn a closed system into an open one.

6 CONCLUSIONS

In our view, the most important contribution of Anatomy of Interaction is the aim to rethink programming and focus on *interaction* rather than *algorithms.* This allows us to see programming from a more human-centric perspective and think about how humans interact with computers and what metaphors allow them to do so.

At the technical level, the paper imagines an *open ecology of function* where systems can be modified by users, components shared and recombined. This poses important technological challenges. In this critique, we focused on the idea of *substrate* that is used to construct such systems. The idea of a substrate lets us visualize the difference between ordinary applications, spreadsheet systems and the system imagined by the authors. In order to build a fully open system, the substrate needs to cover a full range of easy-to-make small changes and difficult-to-make complex modifications. This could be achieved by providing multiple ways of working with the substrate – possibly using multiple metaphors such as *quantum physics* at the low level and *cooking* at the high level.

REFERENCES

[1] Antranig Basman, Philip Tchernavskij, Simon Bates, and Michel Beaudouin-Lafon. 2018. An Anatomy of Interaction: Co-occurrences and Entanglements. In *oceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming’18> Companion).* ACM.
[2] Alan Frank Blackwell. 1998. *Metaphor in diagrams.* Ph.D. Dissertation. University of Cambridge.
[3] Alan F Blackwell and Thomas RG Green. 1999. Does metaphor increase visual language usability?. In *Visual Languages, 1999. Proceedings. 1999 IEEE Symposium on.* IEEE, 246–253.
[4] Georgio Chrysanthakopoulos and Satnam Singh. 2005. An asynchronous messaging library for C#. In *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages.* OOPSLA San Diego, 89–97.
[5] Sylvain Conchon and Fabrice Le Fessant. 1999. Jocaml: Mobile agents for objective-caml. In *Agent systems and applications, 1999 and third international symposium on mobile agents. Proceedings. First international symposium on.* IEEE, 22–29.
[6] Nathan L Ensmenger. 2012. *The computer boys take over: Computers, programmers, and the politics of technical expertise.* Mit Press.
[7] Cédric Fournet and Georges Gonthier. 1996. The Reflexive CHAM and the Join-calculus. In *Proceedings of the Symposium on Principles of Programming Languages (POPL ’96).* ACM, 372–385.
[8] Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc.
[9] George Lakoff and Rafael E Núñez. 2000. Where mathematics comes from: How the embodied mind brings mathematics into being. *AMC* 10 (2000), 12.
[10] Tomas Petricek. 2018. What we talk about when we talk about monads. *The Art, Science, and Engineering of Programming* 2, 12 (2018).
[11] Tomas Petricek and Don Syme. 2011. Joinads: a retargetable control-flow construct for reactive, parallel and concurrent programming. In *International Symposium on Practical Aspects of Declarative Languages.* Springer, 205–219.
[12] Michael Polanyi. 2015. *Personal knowledge: Towards a post-critical philosophy.* University of Chicago Press.
[13] Horst WJ Rittel and Melvin M Webber. 1973. Dilemmas in a general theory of planning. *Policy sciences* 4, 2 (1973), 155–169.
[14] The Eve team. 2018. Eve: Quickstart. *Online at http://docs.witheve.com/v0.3/tutorials/quickstart* (2018).