

Proceedings of the

30th Annual Workshop of the

Psychology of Programming Interest Group (PPIG 2019)

28 - 30 August 2019
Newcastle University, UK



Edited by
Mariana Marasoiu, Luke Church and Lindsay Marshall

PPIG 2019 Call for Papers

Following last year's successful engagement with craft and art, this year we are particularly interested in aspects of interdisciplinarity. The Psychology of Programming is itself an interdisciplinary effort, as well as a multi-disciplinary effort. Previous discussions have substantially engaged with other disciplines, including education, engineering, music, magic, data analysis, collaboration and creativity, and yet there are so many other fields left to consider. Perhaps now is the time to consider the psychology of animal programming?

We also welcome research that discusses the psychology of programming in the wider context, drawing on issues such as sustainability, economy, politics, media, culture and society.

We have open minds, and enjoy conversations around creative and risky ideas more than polished 'correctness'. If you think we might be interested, give us a try.

If you're stuck to think of things we might find interesting, here are some themes to prompt:

- Music(al) programming
- Liveness and interactivity in programming
- Programming education and craft skill acquisition
- Human centered design and evaluation of programming languages, tools and infrastructure
- Programming and human cognition
- Team/co-operative work in programming
- End user programming
- Distributed programming, programming distribution
- Culture and programming
- New paradigms in programming
- Code quality, readability, productivity and re-use
- Mistakes, bugs and errors
- Notational design
- Data programming
- Unconventional interactions and quasi-programming
- Non-human programming
- Technology support for creativity

Looking forwards to seeing you there,

Lindsay, Luke, Mariana

PPIG 2019 Programme & Proceedings Index

Wednesday 28th of August

09:30 - 13:30	Doctoral Consortium	
13:30 - 14:00	Registration with nibbles	
14:00 - 14:15	PPIG Open & Welcome	
14:15 - 15:30	Collaborative Keynote	
15:30 - 16:00	Coffee Break	
16:00 - 17:00	Session 1	
	Coding to Learn and Create: New Modes of Programming for Learners Who Have Been Left Out Colin Clark, Clayton Lewis, Simon Bates and Sepideh Shahi	5
	Beyond a Faster Horse: the UX of a Paperless Biochemistry Laboratory Christopher Martin, Kate Kilgour and Angus Lamond	16
19:00	PPIG Dinner at The Earl of Pitt Street Address: 70 Pitt St, Newcastle upon Tyne NE4 5ST	

Thursday 29th of August

10:00 - 11:00	Session 2	
	Software design as multiple contrasting dialogues Marian Petre, André van der Hoek and David Bowers	22
	Undergraduate students' learning approaches and learning to program Melanie Coles and Keith Phalp	30
11:00 - 11:30	Coffee Break	
11:30 - 13:00	Session 3	
	Parlez-vous Java? Bonjour La Monde != Hello World: Barriers to Programming Language Acquisition for Non-Native English Speakers Brett Becker	40
	Usability of Probabilistic Programming Languages Alan Blackwell, Luke Church, Tobias Kohn, Martin Erwig, James Geddes, Andy Gordon, Maria Gorinova, Atilim Güneş Baydin, Bradley Gram-Hansen, Neil Lawrence, Vikash Mansinghka, Brooks Paige, Tomas Petricek, Diana Robinson, Advait Sarkar and Oliver Strickson	53
	Towards a Consensus about Computational Thinking Skills: Identifying Agreed Dimensions Bostjan Bubnic and Tomaz Kosar	69
13:00 - 14:00	Lunch	
14:00 - 15:30	Session 4	
	Toward meaningful algorithmic music-making for non-programmers Matt Bellingham, Simon Holland and Paul Mulholland	84

Winter is Coding: On Programming, the freeze response, and how design can help
Michael Nagle

The Naturalist's Friend - A case study and blueprint for pluralist data tools and infrastructure 94
Antranig Basman

15:30 - 16:00 Coffee Break

16:00 - 17:00 Panel Discussion

Evaluating Programming Systems Design 106
Jonathan Edwards, Stephen Kell and Tomas Petricek, Luke Church

17:30 - onwards Sightseeing: walk through the city and along the quayside, followed by dinner

Friday 30th of August

10:00 - 11:00 Session 5 - Doctoral Consortium short talks

Clinical Decision Support System Design with Probabilistic Programming Languages
Diana Robinson

Open Piping: a Visual Workflow Environment 117
Charles Boisvert

Constructing a Model of Expert Parallel Programmers' Mental Representations Formed During Parallel Program Comprehension 119
Leah Bidlake

Challenging users' perceptions of decision boundaries in machine learning systems 121
Rob Bowman

11:00 - 11:30 Coffee Break

11:30 - 13:00 Session 6

Probes and Sensors: The Design of Feedback Loops for Usability Improvements 124
Luke Church and Emma Soderberg

Cognitive Dimensions of Modular Noise Improvisation 138
James Noble

Mapping the Landscape of Literate Computing 148
Bjarke Vognstrup Fog and Clemens Nylandsted Klokmose

13:00 - 14:00 Lunch

14:00 - 15:30 PPIG Games

15:30 - 16:00 Coffee Break

16:00 - 17:00 **Keynote**
Ben du Boulay

17:00 - 17:30 PPIG Close

Coding to Learn and Create: New Modes of Programming for Learners Who Have Been Left Out (Work in Progress)

Colin Clark
 Inclusive Design
 Research Centre
 OCAD University
 cclark@ocadu.ca

Sepideh Shahi
 Inclusive Design
 Research Centre
 OCAD University
 sshahi@ocadu.ca

Simon Bates
 Inclusive Design
 Research Centre
 OCAD University
 sbates@ocadu.ca

Clayton Lewis
 University of Colorado Boulder
 clayton.lewis@colorado.edu

Abstract

While learning to code is increasingly becoming mandatory for elementary school students in many countries, learners with disabilities—especially those with complex or intersectional disabilities—are often excluded. These learners depend on assistive technologies to participate in class, communicate with family, and share with their friends. For this reason, we argue that students with disabilities can significantly benefit from the process of learning how to express themselves using computational means, and have the most at stake in becoming producers of technologies rather than simply consumers. Indeed, working with these learners raises significant questions about what coding actually entails, and the motivations and goals for learning how. The *Coding to Learn and Create* project is designing new educational coding tools that support learners with disabilities. With an emphasis on collaborative and artistic activities, we are exploring new forms of programming that support the development of life and learning skills while enabling creative expression and participation.

Introduction

Computers are, of course, everywhere around us today. Software mediates our experience with many aspects of social, school, and work life, yet its inner logics and processes can often be inscrutable to those of us on the outside. Job opportunities continue to grow for software developers and other technical workers, while the increasing role of automation in certain forms of labour, coupled with fears that so-called “artificial intelligence” may impact the role of human agency in many jobs, has invoked a feeling in many that our educational systems may well be unprepared for the future of work (Vilorio 2014, Smith & Anderson 2014). Educators and policymakers have, in response, argued for the need to teach children technical skills such as coding early in their education, as a way to prepare them for employment in an uncertain, technology-driven economy¹. In many cases, these appeals schematically link the value of “learning to code” with “STEM” (science, technology, engineering and mathematics) skills, often with a thin gloss of creativity or art applied on top. Less directly instrumental in its argument (i.e. coding == jobs), but far more wide-reaching in its belief in the importance of purely computational modes of learning is the “computational thinking” movement, which claimed that “computational thinking is a fundamental skill for everyone” and it should be as educationally essential as “reading, writing, and arithmetic... to every child’s analytical ability” (Wing 2006).

With this, perhaps, as the motivating backdrop, coding education has become an increasingly prioritized area of the educational curriculum for young children in many countries. In Canada, where several of the authors live, coding is mandatory in three provinces and has been included in the curriculum in several others. Despite this recent emphasis placed by policymakers, computer scientists, and educators on introducing students to coding at a young age, many learners are excluded from the opportunity to learn due to a lack of accessible tools and instructional methods. Students with disabilities—especially those who

¹ See, for example, the rationales of the CS4All <https://www.csforall.org/> movement in the U.S. and Canada’s CanCode program:

<https://www.canada.ca/en/innovation-science-economic-development/programs/science-technology-partnerships/cancode.html>

have cognitive, communication, or physical disabilities—are often unable to participate in today's classroom coding activities, or are relegated to passive roles while their peers actively engage in solving problems together computationally. Yet these students, who often depend on assistive technologies to participate in class, communicate with family, and share with their friends may have the most to gain from learning how to be creators of their digital worlds, not just passive users. For example, a learner who depends on augmentative and assistive communication (AAC) software may benefit from understanding how communication boards are modelled, created, and installed, and how word prediction algorithms work, so that they can make their own custom vocabularies (e.g. to tell jokes with, or share secrets with their friends but not their parents). Or learners with mobility impairments may be able to use a coding environment connected to a robot to experiment with new forms of agency in the physical world, or learn how to give better instructions to their caretaker.

Working with students with disabilities thus raises interesting questions about what coding actually is as a practice, what it's for, how it is manifested materially, and how it can be productively implicated with other learning and personal activities. By working with learners who are currently on the margins of computational creativity, and addressing the barriers that prevent them from engaging with computational media equally alongside their peers, new possibilities for programming languages and tools may be revealed that benefit all learners—and other programmers and users, too.

The Coding to Learn and Create Project

The Coding to Learn and Create project² (<https://codelearncreate.org>) is aiming to address the barriers to participation in coding education by students with disabilities, particularly those who are most likely to be assumed (incorrectly) to be incapable of coding, such as those with cognitive disabilities (Taylor et al. 2017). The goal of the project is to empower all learners to be creators of their digital worlds, to express themselves using code and art, and to apply these skills to other areas of learning and daily life.

While coding is too often seen instrumentally as a means to develop skills that will support future employment and career opportunities, our approach is to investigate the broader potential for coding to contribute to the development of social, daily living, and creative skills—*coding to learn* (Popat et al. 2019). The argument here is that participation in coding lessons may help support students with complex disabilities especially to develop collaborative and communication skills, strategies for problem solving, task sequencing, spatial awareness, and metacognitive skills such as those involved in giving instructions to others. Participation in coding activities also helps these learners develop a greater sense of belonging and equality with their peers in the school community. We are interested in ways that computational modes of expression can support creativity and learning, rather than simply treating the arts as a secondary concern whose role is largely to make technological concepts more appealing for kids.

The project is establishing an open, evolving repository of inclusive coding resources and activities³. These resources will be released as Open Educational Resources (OERs), and will provide educators with strategies, teaching tools, lesson plans, and techniques to help them teach more inclusively, and to adapt current coding curriculum and programming environments to better match their students' diverse creative, social, and skill development goals. These resources will also include collaborative arts-based programming activities (such as computational drawing, music, and performances) that will provide new ways for students to learn programming while creatively engaging with their peers and the physical environment—

² The Coding to Learn and Create project is led by the Inclusive Design Research Centre at OCAD University with Bridges Canada, and is funded by a grant from Innovation, Science, and Economic Development Canada's Accessible Technology Program.

³ An early version of this *Educator's Toolkit* is available at <https://resources.codelearncreate.org/>

either directly or through avatars such as robots or simulations. This repository of inclusive coding OERs will be open to contribution from others, and will be designed to support reuse and adaptation.

In addition, we are co-creating, with educators, students, and their families, a new programming environment that is designed to support those who are currently unable to effectively use current coding environments. This environment will consider programming as an accessible and collaborative activity in which students may each have their own personalized interface or representation of a program, while also being able to work on shared projects and learning activities together. Additionally, teachers and students will be able to define personalized goals, rewards, and learning scaffolds that will help to support incremental skill development.

To accomplish this, the Coding to Learn and Create project is organizing a series of co-design sessions, collaborative programming workshops, and hackathons to design, test, and refine the project's learning resources and programming tools. These events act as participatory research methods that enable the evaluation, expansion and refinement of the project's deliverables in a collaborative way that is also grounded in the realities of teaching programming to very diverse students, both in the context of inclusive classrooms and congregated, disability-specialized schooling.

The following sections outline two of the key design methods we are employing throughout the project: participatory co-design with students and teachers, and continuous prototyping. We discuss ideas and strategies for how we'll engage learners in the process of creating new coding tools, describe our first prototype of an accessible turtle graphics programming environment, and summarize several areas of computation and creative expression that we will explore with our community.

Co-Designing Accessible Coding

The Motivation for Co-Design

Traditional design practices rarely engage users as participants in a substantive manner throughout the process, particularly those with extreme needs. During the initial discovery phase, researchers may engage users in activities such as interviews, focus groups, and observation sessions to better understand their needs and behaviors in a specific context. Later on in the process, they may also engage people in usability testing sessions and focus groups to get their feedback about solutions that have largely already been built. These efforts, however, do not give users the means to be directly involved in the creation and development of a solution that will inevitably impact them later.

For the Coding to Learn and Create project, we aim to take a different approach, applying a co-design process in order to more actively involve our participants as co-designers within every stage of the research, design and development process. In addition to co-design activities, we will also apply other more conventional research methods, such as surveys and interviews to further expand our reach and gather data that can complement the ideas generated throughout our co-design activities.

Planning for Co-Design

We consider that everyone is creative and capable of contributing to design and problem solving. However, when a creative action takes the form of designing, building, drawing, performing or using other creative media to express ideas, many people think (or have been told) that they do not have the proper skills, expertise, or training to contribute to such activities. This becomes even more challenging when our co-designers are from marginalized populations, such as those with different learning, cognitive and physical needs. They may feel uncomfortable sharing their ideas or be afraid of contributing thoughts that might be perceived as less good, not right, inappropriate, or irrelevant. To minimize this barrier, we will start with small-scale activities, encouraging students to participate in tractable problem-solving tasks, building up trust, and inviting them to experiment with different ways of contributing in order to find their favorite

medium. This approach helps them develop confidence and gives them—and us—ample opportunities for experimentation, determination of roles and modes of participation, and time to fine-tune the collaborative problem-solving process together.

The other aspect that may impede participation is the accessibility of the medium. Even the most confident individuals may not be able to fully participate in a co-design process if they are not provided with tools and activities that meet their needs. For example, learners with disabilities may require different ways of accessing and communicating information, such as via voice commands, gestural commands, eye gaze, communication boards, single switches or other assistive devices to be able to communicate with each other. Thus, without access to such assistive technologies—and materials that are compatible with them—they won't be able to fully participate. To this end, we are endeavouring to ensure that our tools and activities are multimodal and support a wide range of input and output methods, and are working closely with the students and their care providers to determine the appropriate tools and formats.

Co-designing with people who have cognitive, learning, or communication disabilities can be fraught with unique challenges. Depending on the context and the participants' needs, there will be situations where a student will not be able to independently participate in the co-design process, and may require assistance from or mediation by their caregivers. This may impact a student's agency and can raise questions about how truly involved they are in a co-design process. To address this issue, our team will work closely with the care providers to ensure students' insights are captured as they prefer and try to avoid overly interpreting their ideas. This requires a different approach to facilitation and engagement. For example, we have observed that inexperienced facilitators or assistants have a tendency, when asking questions of students with communication disabilities, to rush or anticipate the student's responses. Some students need more time to consider a question and respond to it, and the task of clarifying or interpreting a student's response often requires further questions and prompts. During our co-design engagements, we will provide educators and their assistants with communication strategies that give students sufficient time to think about their responses and avoid rushing them to respond or asking leading questions. This may involve tailoring the length of a session, or reducing the number of topics discussed to a more tractable scope. In addition, during the sessions themselves, we will look out for opportunities to provide students with on-the-fly adaptations that may be a better fit for their direct participation, and which will minimize the risk of misinterpreting their ideas.

At this early phase of the project, we are in the midst of reaching out to different communities and building relationships with various schools and organizations that work with students with complex learning, cognitive and physical needs. In the meantime, through collaboration with our initial group of partners and contributors, we are developing a “palette” of different co-design activities that will suit different contributors and situations, and which can be further refined as we work more closely with our co-designers and their care networks. Some of these activities are described in the next section, where we outline a process for working with students and teachers to build a more accessible and educational coding tool.

A Process for Discovering What Kids Want to Learn

To design and develop an accessible educational coding environments for kids, first we need to reassess our assumptions about coding—what it is, and how it should be taught to learners with diverse needs—and to start our work from their goals, interests, and perspectives. We will specifically recruit learners who have been marginalized or excluded from coding education due to their different needs or abilities, who will have an opportunity to share their experiences and express their visions of coding and how it could impact their lives.

Once we break out of our unquestioned assumptions about coding and consider alternative models for existing systems, we can start building new coding tools and resources. At this phase, we will work with our co-designers to design, build, and test various coding environment prototypes and tools to help actualize different models of coding that are multimodal and accessible for a wide range of needs.

Teachers and educational assistants who have relevant experience in coding will be involved throughout this process in different capacities. Since they are familiar with their students' needs and preferences, we will work with teachers directly to ensure our co-design activities and tools are accessible for all participating students. Teachers will also play an instrumental role in helping us to understand curriculum requirements, educational goals, and the practical, day-to-day challenges teaching coding to students with disabilities. They will also help bridge the communication gap and help our team to work with their students through the co-design process.

At each phase of the co-design, we are planning to use a series of individual and collaborative activities in order to engage students in a reflection and discovery process about their perception of coding, and to involve them in generating ideas that will support the design of new coding tools. These hands-on, interactive and multi-modal activities involve a mix of individual and small group participation, providing each student a chance to think on their own before collaborating with their peers. These activities build on each other to gradually introduce the students to the tools and processes used in our co-design process. The insights and ideas generated through these activities will help our team to better understand how students perceive computers and computational processes and how they prefer to learn about and engage with these processes.

The first activity in this series invites students to identify the computers they encounter in their day to day life. They are encouraged to think beyond the stereotypical concept of computers and try to identify any other forms of computation that have an impact on their lives.

A follow-up activity, inspired by Judy Robertson's work in classrooms (Robertson 2019), aims to further investigate the students' understanding of computers and how they perceive computation. Thus, they are tasked with imagining and expressing how a computer works. During this activity, students may touch upon different concepts related to coding and computational processes from their perspective.

This activity will engage students in an ideation process. Here, we encourage them to think about how they would most like to interact with those different computers they have identified in their lives. They will be provided with several challenging scenarios to help them think outside of their ordinary relationships with computers and consider some of the accessibility challenges that coders may face. For example, some of these scenarios may include giving commands to a computer that can't hear the student/a computer that can't see the student/ a computer that is out of their reach or even invisible to them, etc.

In the last activity, students are encouraged to work together to determine how they prefer to learn about computers and interacting with them. In this activity they are asked to be critical and to discuss what is not working in their current coding classes, what could be changed or improved and how they envision the future of coding education.

Once our co-designers have participated in these activities and feel more comfortable with working in a collaborative setting, we will introduce them to more advanced activities that involve interactive coding tools. At this stage, students will have a chance to:

- try out the different prototypes we are designing and developing
- sketch different user scenarios based on their personal experiences with computers to test different aspects of each prototype and identify their accessibility barriers
- share ideas about how they can change, improve, or redesign the prototypes in a way that suits them best. Students are encouraged to document their reasoning for each suggestion, and to imagine changes they would make to the prototypes.

The insights and ideas generated through these activities will help our team to better understand how students perceive computers and computational processes and how they prefer to learn about these processes. To better identify the gaps in the current computer science and coding education for kids, we are also planning to work with teachers, particularly special education educators with relevant experience.

In addition to our planned co-design engagements, we will also reach out to a broader range of teachers across Canada using a survey that has been designed and distributed to more than a hundred schools. In this survey, teachers have an opportunity to discuss challenges that they face with regard to coding education, working with students who have special needs, and opportunities they see for improving or revamping the current coding tools and curricular activities designed for their students. The survey participants can opt-in for an interview to further discuss their perspectives about a more inclusive and accessible coding education.

Continuous Prototyping as a Co-Design Method

We have started work on a prototype of a new coding environment. This prototype is not the thing that we are ultimately building, but rather, a thing to help us make the things that we will co-design with our community of students, educators, and families. Building an early prototype gives us a concrete platform on which to try out design ideas and technical approaches, and helps us when talking with others about the potential approaches and directions of the project. It serves as an invitation to participate—“this is the sort of thing that we want to build with you.” In this way, we have aimed to build something that sits somewhere between a design sketch and a product. Functional enough that we can try real interactions out (such as having a group of students program a robot to make art), but not so set that it limits our conceptions about what a coding environment could be.

Coding to Learn and Create Prototype

Commands

Program

Program:

forward left forward left forward left
 forward left

Available actions: forward, left, right.
 Write your program actions in the box above and separate them by spaces. For example, to draw a square, use: [forward left forward left forward left](#)

[Run](#) [Step](#) [Restart](#)

[Connect to Dash](#)

[Connect to Sphero](#)

Figure 1: The Coding to Learn and Create Prototype. <https://prototype.codelearncreate.org/>

The prototype is built with web technologies and the Infusion JavaScript framework (Basman, Lewis, Clark 2015). This technology platform was selected for the prototype to enable ideas to be implemented quickly and to enable us to run the prototype on the wide range of hardware devices for which web browsers are available and common in educational settings (including desktop computers, laptops, tablets, and Chromebooks). In addition to running on multiple hardware devices, web technologies have well-supported assistive technology integration and some built-in user control of web page presentation (such as zooming or setting of text size). The prototype is architected as a composition of loosely coupled parts, using Infusion’s “Inversion of Control” and “Model Relay” supports. Features of the prototype, such as editors, or robot integrations, can be added, removed, or replaced in a straightforward manner, with minimal coupling to other parts of the system. Enabling the convenient implementation of alternatives and the replacement of parts as the prototype is evolved.

As we work with educators, students, and families to co-design the new coding tools, we will assess the technology choices that we have made for the prototype and select technology platforms for the end product development that best fit the needs of our community, ensuring that what we build will work on the devices and integrate with the technologies that students are using.

Turtle Graphics and Robots

The prototype currently provides a simple turtle graphics system and a simple language consisting of three operations (currently called “actions” in the environment): move forward, turn 90 degrees to the left, and turn 90 degrees to the right. We have chosen to begin with a turtle graphics model due to its familiarity, potential for making visual art, historical usage in coding education, and compatibility with commonly available robots.

In addition to drawing a turtle on the screen, we have implemented integrations with several off-the-shelf coding robots, including the Dash robot from Wonder Workshop and the Sphero robot. Students can connect up to one or each of these robots to the prototype and when a program is run, the robot(s) will move accordingly. The robot integrations are built using the Web Bluetooth API (Web Bluetooth specification). A “Sketch Kit” add-on is available for the Dash robot that enables the attaching of a pen to Dash. With this attachment, a student can use the prototype to make drawings on paper. We are continuing to explore collaborations with robot manufacturers, with the goal of enabling our software to work with the different robots that are available for purchase and which are being used by students and in schools.

Concurrent Notations

One of the areas that we would like to explore with the project is providing multiple concurrent programming notations. Eventually, this will allow a student (or educator or family member) to select the notation, or notations, that are best for their learning needs—while still being able to collaborate with students who may prefer a different notation. In the current prototype, we offer two notations. \

The first notation we implemented is a simple text notation, where a program consists of a list of words, separated by whitespace (where each word is one of the available actions “forward”, “left”, and “right”). For example, to draw a square, the following program could be used “forward left forward left forward left forward left”. The second notation is a symbolic notation consisting of a sequence of symbol blocks, one symbol block per action. The user interface for working with the symbolic notation consists of a palette of actions (move forward, turn left, and turn right), program manipulation commands (insert space into the program, and delete a step), and a program area. Although drag and drop is typically used in block-based programming environments, this user interface pattern can be problematic for some students with physical disabilities. Rather than using drag and drop for building programs with the symbol blocks, we chose to implement an alternative two-step interface similar to the “hybrid method” described in (Milne and Ladner 2018): step 1) select the action to be performed in the palette and step 2) select the target within the program to apply the action to. We have designed the interface to have large targets (symbol program blocks and buttons) to ensure that they can be used without needing very fine motor control, or by those using assistive technologies such as eye gaze control.

Our prototype programming environment has a single underlying program model, currently consisting simply of a JavaScript array of words. Each notation is implemented as a bi-directional mapping to and from the shared program model. This enables the student to work in either notation and have the other be updated automatically. In the future, our programming environment will provide programming functionality beyond a simple sequence of actions, such as subroutines, looping, and variables. When these features are added, our programming model will of course need to become more complex to accommodate this functionality. There are many notations that we would like to try out, and the questions of a) which notations and programming models are most useful to our students and b) how to, or if we can, implement useful mappings between these notations will be topics that we will be tackling over the course of the project.

Some candidate notations for exploration include:

- An interface for controlling a robot, or turtle on-screen, that can act both as a direct ‘remote control’ and a programming environment. Where actions, such as movement, maintain their ‘liveness’ and can be used both to send commands directly to a robot and within a program.
- An interface that facilitates both direct manipulation and programming. For example, where a student could draw a shape directly, rather than with movement commands and then process their drawing programmatically, such as to repeat it with variation (for example with variation in position, rotation, colour, or line thickness). Or where programming commands are derived from the drawn shapes.
- An interface that combines programming instructions with physical control input devices. Where, for example, a parameter could be controlled by pushing or squeezing a ball.
- Dataflow programming as an alternative to imperative programming, where values within a system are related to one-another through connections and transformations.
- Notations for time-based media such as animations, music, and controlling of robots (for example a dance performance).
- Liveness within the programming environment, for example where a program is continuously running and updates to the program take effect immediately (rather than having to press a “Run” button).

Further Topics to Explore

In developing our co-design activities, there are a number of other matters we hope to explore with teachers and students, and to implement in our new educational programming environment.

Connections between coding and art and music.

One of the schools with which we are working has had great success in engaging students with physical and cognitive disabilities in art activities, some producing extraordinary results. At Beverly Public School in Toronto, artists Hien Quach and Patrick Moore developed a series of art creation processes that were personalized to the needs and abilities of their students. These processes were applied by the art class to create a series of large-scale mixed media canvases that were exhibited at the Coding to Learn and Create project launch event. These art projects suggest a potentially germane link between art and computation, where the students can learn about, perform, and create their own formalized processes and instructions for creating art in digital or physical media using code.



Figures 2 and 3: Process-based artwork created by the students at Beverly Public School.

We hope to link coding to other art activities, using the turtle graphics prototype as a starting point for some. The experience of artist Jim Johnson using a form of turtle graphics (see discussion in Repenning et al., 1998) provides inspiration. The turtle Johnson used was able to respond to lines as well as draw them, a possibility we may wish to explore.

Much has been done in producing music under program control, and we hope to explore this medium as well. Here we may be able to draw on some of our own work (#Flocking, Nexus), adapting the systems in response to the experiences of and with students in the new project.

More elements of computation.

A number of extensions to the core design of our turtle graphics prototype can be seen in prior systems, and serve to enrich the exposure to computational ideas that these systems provide. We expect to add a facility for saving and reusing sequences of commands, thus providing a simple form of subroutine. If we add the ability to give the saved sequences names we provide a form of procedural abstraction. Other extensions can provide arguments for some operations, such as a repeat action for which a number of repetitions can be specified. One can then move to support parameters for user-defined subroutines, and further to allow values calculated and stored in variables to be used as arguments. As with other aspects of the work, how much or little we explore in these directions will depend on the response of the students, both with respect to what they want to do, and with respect to what they understand and adopt.

Further abstractions.

Arguably there are other ideas about computing that are important in understanding its pervasive role in the world, but that may not play much part in practical work. One idea is that a single kind of data, numbers, or bits, can represent indefinitely many, very different kinds of things. Another is that a relatively small set of quite simple operations, on numbers or bits, that is, machine instructions, serves to emulate an enormous range of activities. Certainly the role of computers would be very different, and far more limited, if these ideas did not work out. Will our students be interested in these ideas? Can we develop the ideas in an engaging and clear way?

Back to the arts.

Moving back from computation as we know it, we think that some further computational ideas, or aspects of them, can be exercised within artistic expression. An inspiration is the loop machine, a simple device that records and plays back sounds, in such a way that a complex performance can be built up in layers, with later sounds added to ones already recorded. The loop machine demonstrates the ability to store and retrieve information, in a concrete way, while enabling new kinds of musical performance. We think it may be possible to extend this idea into the visual domain, allowing drawings to be captured, replayed, and mixed. Parallels between the two kinds of loopers may help to communicate parts of some of the ideas above, about how computers can represent and operate on different kinds of information in similar ways.

Conclusion

The Coding to Learn and Create project is designing new educational coding tools and teaching resources to support students who have been left out of learning how to code. With a particular emphasis on students with complex, intersectional disabilities such as cognitive, learning, and physical disabilities, the project

aims to reconsider the goals, motivations, and materials of how software creation is taught in light of the unique learning, creativity, and communication needs of these students. By recentring coding as a means for supporting art, life skills development, and expressive communication, the project aims to support learners with disabilities in engaging creatively and collaboratively as producers of computational media. The project's approach is rooted in community-based, open source co-design practices in which educators and students will have a significant voice in shaping the direction and outcomes that we create. We welcome participants from a broad range of backgrounds, including computer scientists, artists, educators, and researchers.

References

- Basman, Antranig, Colin Clark, and Clayton Lewis. "Harmonious Authorship from Different Representations (Work in Progress)." In *Proc. PPIG 2015 Psychology of Programming Annual Conference. Bournemouth, England, 15th-17th July*. 2015. [[PDF](#)]
- Milne, Lauren R., and Richard E. Ladner. 2018. *Blocks4All: Overcoming Accessibility Barriers to Blocks Programming for Children with Visual Impairments*. In Proceedings of the ACM Conference on human factors in computing systems (CHI '18). ACM, New York, NY, USA. [[PDF](#)]
- Popat, Shahira, and Louise Starkey. "Learning to code or coding to learn? A systematic review." *Computers & Education* 128 (2019): 365-376.
- Repenning, A., Ioannidou, A., & Ambach, J. (1998). Learn to communicate and communicate to learn. *Journal of Interactive Media in Education*, 1998(2). [[PDF](#)]
- Robertson, Judy. "Answering Children's Questions About Computers." *Communications of the ACM*, January 2019, Vol. 62 No. 1, Pages 8-9. <https://cacm.acm.org/magazines/2019/1/233512-answering-childrens-questions-about-computers/fulltext>
- Smith, Aaron, and Janna Anderson. "AI, Robotics, and the Future of Jobs." *Pew Research Center* 6 (2014).
- Taylor, Matthew S., Eleazar Vasquez, and Claire Donehower. "Computer programming with early elementary students with Down syndrome." *Journal of Special Education Technology* 32, no. 3 (2017): 149-159.
- Vilorio, Dennis. "STEM 101: Intro to tomorrow's jobs." *Occupational Outlook Quarterly* 58, no. 1 (2014): 2-12.
- Web Bluetooth specification, Web Bluetooth Community Group <https://webbluetoothcg.github.io/web-bluetooth/>
- Wing, Jeannette M. "Computational thinking." *Communications of the ACM* 49, no. 3 (2006): 33-35.

Beyond a Faster Horse: the UX of a Paperless Biochemistry Laboratory

Chris Martin

Lamond Lab

University of Dundee

crmartin@dundee.ac.uk

Kate Kilgour

Lamond Lab

University of Dundee

k.j.kilgour@dundee.ac.uk

Angus I. Lamond

Lamond Lab

University of Dundee

a.i.lamond@dundee.ac.uk

Abstract

It is astounding though possibly not surprising, that the default cognitive prosthesis in the modern laboratory environment is the paper notebook. In many walks of life, the 50+ year-old promises of technology are increasingly a reality: spoken dialogue systems a commodity, central-heating systems that can anticipate need and context-aware delivery of advertising as you walk past a shop. With all of this capability, why are paper notebooks still the best option for many working in science laboratories? This paper describes a study designed to try and understand why paper remains prevalent. It seeks to understand what feature set and design decisions are required to inform the design of an Electronic Lab Notebook (ELN) capable of displacing the paper notebook.

1. Introduction

Henry Ford famously posited: “*If I had asked them what they wanted, they would have said a faster horse*”. The type of transformation to personal transportation brought about by the motor car is in some ways similar to what we seek to achieve with the electronic notebook (ELN). We are not simply looking for a digitised version of what scientists currently do in a note book, One Note or Evernote already provide this and are used with mixed success. We are looking for a transformative tool that understands the array of stakeholders engaged in science and the complex environment and geography where this work takes place. The challenge of UX is not to simply ask users what they want, rather it is to work with them to understand what they need. This work in progress paper is arranged as follows. First, the context for the ELN will be described, outlining the locations and range of equipment used, by whom and crucially what type of tasks need to be supported and documented. This paper will then go on to describe the approaches proposed to capture stakeholder insights. Finally, some preliminary finding shall be presented.

1. Many ordinary users in an extraordinary environment

A biochemistry laboratory is a nuanced, complex and sometimes hostile environment. Depending on the task at hand, there may be a range of health and safety requirements that can inhibit the typical user’s abilities. At a minimum, a user will employ gloves, eye protection, and a lab coat. Depending on the substances used, they may require to work in a fume hood. This is an enclosed compartment with a sliding-front window, or sash, that can be drawn down to prevent contaminants entering the working area, or conversely, protect against toxic fume exposure or chemical spills. In some cases, these precautions are in place to protect scientists, though often these precautions aim to preserve the integrity of the experimental sample. It is also common for areas to be dedicated for the sole purpose of a single activity, such as tissue culture. Therefore, it is likely that the execution of an experiment will take place in a range of locations which are dependent on the particular action being performed.

This environment is peppered with equipment; in some cases, it will be for the sole use of an individual, e.g. microfuges, shakers, heat blocks or pipettes. Where equipment is more expensive, or perhaps used intermittently, it will be shared with others. This may be a resource for dedicated laboratory use, or for use within a wider department or an entire institution. In addition to the equipment used to manipulate and interact with experimental samples, there will be a range of plasticware consumables such as pipette tips, tubes, columns, and tube racks. There will also be various chemicals and biochemicals used at different stages of the experiment. Some of these items will be stored at room temperature on shelves and many will be stored in refrigerators or freezers. It is typical for an individual scientist to have one, or more, personal refrigerator and freezers. The owning scientist can then use these to store personal stocks of everyday items or experimental samples they are currently working with. Other items may be stored in communal locations within a laboratory space. For long term storage, there will be institution-wide resources that can be utilised.

The laboratory is inhabited by a range of intelligent, driven individuals that must execute experiments methodically as they interact with the wide range of complex instruments, equipment, chemicals and bio-chemical stock described previously. There will likely be an array of different stakeholders in a laboratory environment. This can include principal investigators, postdoctoral researchers, Ph.D. students and undergraduate students. Additionally, in some cases, there also may be support technicians and laboratory managers. Although this network of stakeholders strive to fulfil a common, overarching goal of delivering insights via high-quality science, their individual contributions will vary and thus the supporting tools must fit a range of user needs. Occasionally, there will be overlapping and complementary goals, and in other cases goals may conflict.

Science is a highly collaborative environment, not only within laboratory groups but across different geographic sites. Modern scientific studies tend to employ a multifaceted approach requiring a wide range of expertise often found across multiple institutes. Currently, a lot of computer-supported collaborative work is supported via fairly primitive tools such as email and online file stores. Perhaps unsurprisingly, spreadsheets feature heavily. This is a good example of a single-use data repository; a spreadsheet manifest which describes the contents of a sample delivery is disposable and has a very short half-life. Once the delivery is received, the manifest file is of little value and will most likely be archived or discarded.

The design, execution and reporting of an experiment will happen across a range of locations, over an extended time period and with various stages requiring different types of support. The collection of stakeholders described will perform different roles in supporting these tasks. Each role will have different expectations and require different levels and types of assistance to produce good science. The cornerstones of good science are robustness, reliability reproducibility. We can assist in forming a foundation for this through effortless access to experimental designs and meticulous capture of experimental metadata. The current approach with personal, paper-based notebooks only serves to support the individual in isolation. This is not at all surprising, as the notebook is a cognitive prosthesis for the individual as they discharge their duties. We propose that, with a rich understanding of the interconnected web of stakeholders, it is possible to design a system that can provide a highly flexible, personal support tool that can also inform the wider challenge of collaboration, research project-management and facility management.

In summary, our problem space is high dimensional and complex. We have a range of different stakeholders contributing different effort to a central goal. The physical environment is varied and presents some significant design challenges. Within this environment, there is a vast array of specialist equipment and consumables, all utilised to perform high-quality scientific research which must be consistently executed and meticulously documented. Projects can have durations varying from months to years and often involved collaborator's spanning different geographic locations. This study aims to improve the understanding of the work the scientist performs, discover the everyday tools and technologies they currently employ and explore the different priorities of various stakeholder groups to inform features required of an ELN.

2. Study design

Although the software development team has over 10 years of experience delivering software solutions for scientists, we were keen not to limit our findings by our experience and assumptions. To that end, this study has been designed to be open and unbounded but adheres to many of the characteristic of a Semi-structured qualitative study (Blanford, 2013). User time is a valued commodity and to ensure maximum return, a phased focus group (Gill et al, 2008) was devised to address the following three questions:

1. What tasks make up a typical day of work for a biochemical scientist?
2. What tools and technologies are routinely used to support the tasks described in Q1?
3. What are the scientists priorities for an ELN?

Participants for the pilot were recruited from the Lamond Lab group and verbally consented at the beginning of the session. Participants were guided through three tasks addressing each of these research questions. A facilitator introduced each task and offered points of clarity and rationale. The study had

a purposeful, well-defined structure, however the intention was always to capture core data whilst seeding relevant, reflective discussion within the group. The session was audio recorded and a second facilitator was also present to take notes as well as prompting the group with regards to timings, etc.

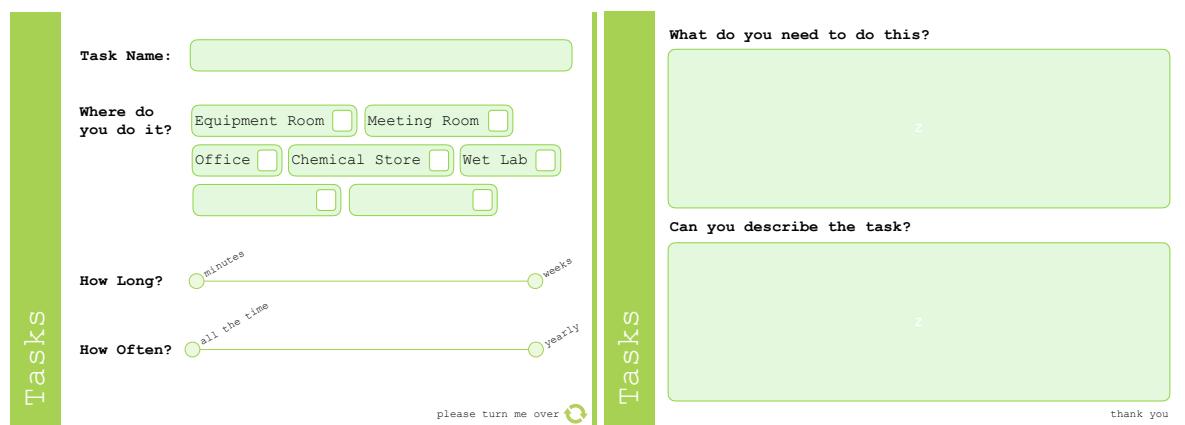


Figure 1 – Task postcard, front and rear.

Questions 1) and 2) were addressed using a lightweight, staged, survey card approach. To address question 1), participants were given a stack of custom-designed postcards where they could fill out the task's name, location in which it's performed, duration and frequency. This was intended to be a quick-fire activity. To reduce participant effort and encourage a flow state (Nakamura & Csikszentmihalyi, 2014), the postcard (fig 1) design offered checkboxes with common locations and visual analogue scales (Krosnick & Fabrigar, 1997) representing duration and frequency. When the completion of cards by participants came to a natural rest, the facilitator directed participants to turn their cards over, where a small free-text box asks participants to provide a brief description of the task and indicate items that are necessary to perform the task. This takes a little more effort. Upon completion of these postcards, each participant is encouraged to identify their most frustrating task and share this with the group to seed a wider discussion. To address question 2), the front face of the card captured identical metadata to question 1), whereas the reverse face of the card requested a description of the tool's use with space to list associated pros and cons. When participants had completed the support-tool cards, they were asked to order them from most useful tool to least useful tool. Each participant was asked to describe their most indispensable tool.

To address question 3), a closed set of 13 terms, pertinent to ELN, was generated. As a group, the facilitator leads a collaborative insertion-sort of the terms. When a new card is presented, the term is described by the facilitator and a shared understanding of its meaning is agreed on by the group. The new card's importance is then discussed in relation to each existing card in the list, generating a prioritised list. The purpose of this exercise is threefold. Firstly, to ensure the development and design teams understand the key vocabulary as defined by the users. Secondly, to create a prioritised list of important features that can directly feed into our development process as we begin construction of our product. In future, It will be interesting to observe the differences, if any, there are between stakeholder groups. Thirdly and finally, performing this exercise as a group was a conscious design decision to encourage externalised reasoning. If there is a split of opinion in the group, each party must articulate their argument for and against. This type of collaborative reasoning also helps users understand the compromises that must be made as part of the development process.

3. Preliminary findings

The study design described was piloted with six scientists from the Lamond Laboratory based in the Centre for Gene Regulation and Expression within the School of Life Sciences at the University of Dundee. These scientists were all experienced, postdoctoral researchers who were working on various projects in a subfield of molecular biology known as proteomics. The session lasted for 1 hour 22 minutes and there were no deviations from the design described previously.

3.1. Task card findings

For the initial exercise, 30 task cards were completed. These cards were sorted post hoc by the facilitator into five emergent themes. **Sample Preparation** was described on 16 cards, with a wide range of noted durations, from one hour to three days; these activities are performed frequently and exclusively in the “wet” laboratory. **Sample Processing** - processing samples on the mass spectrometry (MS) instruments - was the second-most reported task with six completed cards; the duration of these tasks ranged from two hours to two weeks and they were also reported as frequently occurring, unsurprisingly occurring exclusively in the instrument room, where the MS instruments are housed. **Tissue Culture** was the third-most reported task with three completed cards where durations ranged from 30 minutes to one day, occurring anywhere from daily to monthly. This activity happens in a separate, secure “wet” laboratory space which provides a stronger degree of control over air flow, etc. **Data Analysis** was the fourth-most reported task with three cards with durations from three hours to multiple weeks. This activity is discharged in the office space generally involves a range of domain-specific software packages which process the raw files generated by the MS instruments. **Literature Review** was also reported on one card, with a duration of three hours at a weekly frequency.

There was some consensus in the group around tasks that frustrate; these included activities that are perceived to waste time, failed quality controls, instrumentation/equipment failures and external dependencies such as engineer call-outs. MS systems are incredibly fragile and require regular, preventative maintenance and cleaning. The task cards and discussion indicated that a large portion of a scientist’s time is spent doing bench work, performing experimental steps using equipment and consumables.

3.2. Supporting tool card findings

For the second exercise, 18 tool cards were completed and sorted into another five emergent themes. **Domain-Specific Analysis Software** was reported on five cards, extending in duration from minutes to days. These tools were reported as being employed frequently to weekly, occurring in the office. **Simple Bench Kit** was reported on four cards, with a frequency of constant use in the “wet”-laboratory location used for durations ranging from minutes to hours. **Process Tracking** was reported on four cards with a frequency of constant use, durations between 1-20 minutes and with dual locations of office and instrument room. Many participants reported using paper notebooks for process tracking. Flexibility and portability were noted as important, positive aspects of paper notes and several participants reported that the act of handwriting their notes assisted with memory retention. An interesting duality emerged whereby many scientists keep a “scratch” notebook where rough notes and ideas were captured, which were used to inform formal notes in a paper laboratory notebook, Microsoft Word document or similar. **Non-Domain Specific** software, predominantly Microsoft Excel, was reported on four cards with a frequency of constant use and duration ranging from minutes to hours. The positive attributes for Microsoft Excel were reported as its ease-of-use, flexibility and wide range of accessibility to many different stakeholders. The final tool noted was **Internet Journals**, used every day from minutes to hours.

The flexibility and portability of paper notebooks will always be hard to compete with. The added value of an ELN will come from opportunities which integrate additional, required support tools. The flexibility of Microsoft Excel is tempered by the duplication required to repeat routine tasks. Arguably, where a routine task is consistently performed in Microsoft Excel, e.g. calculating volume required for desired solution concentration, it should be possible to reduce the work done by the user to initialise this type of spreadsheet calculation. In addition to reducing the effort required from the individual, the likelihood of an error occurring is also reduced. Tools like Microsoft Excel can be regarded as unmanaged and therefore any audit trail is very much at the discretion of the user, dependent upon their personal habits of file management such as organisation and backup. An additional opportunity to reduce the scientist’s work, whilst simultaneously improving metadata capture, is to model all aspects of their scientific process. If an experiment requires a specific reagent and this reagent is modelled in the system, then there is no need for the scientist to manually note the batch numbers, etc. The user can simply form a digital association with an existing stock item, using a pick list or even a barcode attached to the physical stock item.

3.3. Priority sort findings

The final task of insertion card sorting stimulated a good deal of conversation. This was the final task performed and provided a good opportunity for group discussion. The agreed upon final list of ordered cards is presented in table 1, with accompanying remarks.

Position	Term	Remarks
1	Security	No ambiguity or surprise that this is paramount
2	History/Audit Trail	Immutability is important as “crossing out” of mistakes needs to be transparent and logged, much like a legal document.
3	Calculations	Calculations are done very frequently, on the fly at the bench. Smartphones are very frequently used as calculators.
4	Protocol (SOP in dev)	Access to the software on mobile device described as essential. Would be ideal for dictating verbal notes, scanning barcodes, taking pictures etc.
5	Standard Operating Procedure	
6	Portability	Access to the software on mobile device described as essential. Would be ideal for dictating verbal notes, scanning barcodes, taking pictures etc.
7	Location Management	
8	Notifications	Notifications are perceived as being more of a hindrance than a help. Very negative reaction to the idea of receiving notifications, other than those from personal timers. Otherwise, a very useful exception for notifications would be MS alerts for calibration failures.
9	Sharing	Choosing which users can have access/share your experimental data is very useful.

Table 2 – Card Sort.

The term **Standard Operating Procedure** (SOP) was disambiguated with **Protocol**. Where a protocol is an experimental design in-progress, which may be used numerous times, being tweaked and modified as per the scientist's desires. In contrast, an SOP is regarded as a fixed experimental design that should not be deviated from. In the context of academic research, a facility's protocols are far more prevalent than SOPs.

The term **Notification** evoked a strong negative response from the majority of participants. There was a strong feeling that notifications were an unwelcome distraction and intrusion into their work. This topic was explored further by discussing the extent to which participants tuned and managed notifications they receive on their personal smartphones. For instance, Facebook notifications may be turned off, SMS messages may be visible but not audible, and calls from certain numbers may have an associated ringtone compared to unknown numbers. The range of notification configurations that participants had on their smartphone's was quite interesting and was used as an analogy for the *ELN* app. The participants then spent some time thinking about information they would want to know about in the form of a push notification. For example, if an instrument was booked for in a week's time, but has since developed a fault. Or, when a crucial reagent which is close to expiration. The important lessons learned were that “push notifications” is predominantly an engineering term. When presented to the user, it requires a few leaps to determine the value it may add to them. This illustrates that potentially valuable features run the risk of being dismissed as it lacks an interpretable, obvious value for those in the room.

4. Conclusions

The intention was to design an engagement exercise that could be delivered in one hour to a wide range of stakeholders and deliver insight into the daily life of a laboratory-based scientist. This exercise needed to answer simple questions relating to routine tasks and the technology employed to support them. Despite there being an existing relationship between participants and participants being in relatively small number, new information was certainly obtained. The run-time of the exercise was possibly longer due to this familiarity within the group and various informal chat which was interspersed. It is, however, important to establish a rapport with the participants to ensure a rich dialogue. Moving forward, we intend to recruit further participants representative of the various stakeholders in the described problem-domain. This will enable us to methodically broaden our understanding if the rich network of stakeholders engaged in life science research.

5. References

- Gill, P., Stewart, K., Treasure, E., & Chadwick, B. (2008). Methods of data collection in qualitative research: interviews and focus groups. *British dental journal*, 204(6), 291.
- Blandford, A. E. (2013). Semi-structured qualitative studies. Interaction Design Foundation.
- Nakamura, J., & Csikszentmihalyi, M. (2014). The concept of flow. In *Flow and the foundations of positive psychology* (pp. 239-263). Springer, Dordrecht.
- Krosnick, J. A., & Fabrigar, L. R. (1997). Designing rating scales for effective measurement in surveys. *Survey measurement and process quality*, 141-164.

Software design as multiple contrasting dialogues

Marian Petre
 Open University
 m.petre@open.ac.uk

André van der Hoek
 University of California,
 Irvine
 andre@ics.uci.edu

David Bowers
 Open University
 David.bowers@open.ac.uk

Abstract

Software design is a complex pursuit – technically, cognitively, and socially. Understanding that complexity – and managing it effectively – are ongoing challenges. Building on decades of empirical research on professional software design, and on existing literature, this paper presents a new characterization of software design that unpacks that complexity. The characterization drills down to the core of design as a goal-driven activity and expresses it in terms of parallel contrasting dialogues: (1) a dialogue between problem and solution, (2) a dialogue across application, interaction, architecture, and implementation design, (3) a dialogue across the design cycle of analysis, synthesis, and evaluation, (4) a dialogue between pragmatism and fitness-for-purpose, and (5) dialogues among the team members engaged in the work of designing,. There is an inherent tension and interaction between the dialogues, which emphasise different views. This is a mechanism by which effective designers manage the complexity: each dialogue provides a focus (if not a simplification) for design reasoning, but effective design maintains the interaction between dialogues and makes use of the contrasts between them to achieve design insight. This characterisation helps to explain both why existing software engineering methodology does not always work and what an effective ‘design mindset’ is; the paper discusses some of the implications of viewing current software design practices in this light.

1. Introduction

Software design is a complex pursuit – technically, cognitively, and socially.

We start with a definition of ‘design’: to decide upon a plan for a novel change in the world that, when realized, satisfies stakeholders as fit for purpose. This is a useful definition that captures three key characteristics: novel change, context in the world, and fitness for purpose.

The technical landscape changes continually, with shifts in technology, scale, and focus. The technology is used to address problems in multiple domains, and to satisfy multiple stakeholders with different expectations. Design is conducted in different social contexts, and at all stages of software development, from greenfield design to product lines to maintenance.

Software developers tend to be clever people, but nevertheless often software is late, it doesn’t meet its specification, it doesn’t work properly – why? And why, in contrast, are there nevertheless individuals and teams with exceptional track records, who repeatedly deliver software on time, under budget, working first time? In a landscape where technologies and infrastructures change orders of magnitude faster than personnel, one thing remains of constant importance: the ability of developers to be great designers. So what exactly sets expert software designers apart, and what enables them to achieve repeated and enduring design and development success, regardless of the technology or infrastructure of the moment?

The introduction of software engineering and methodology was historically a response to the ‘Software Crisis’: the increasing need for increasingly complex software, without a population of good developers at the ready to produce it [Haigh, 2010]. As Petre and Damian [2014] argued, software development methodology is about systematising (process) and standardizing (process and outputs) in order to achieve consistency and thereby provide leverage for communication, coordination, and, notionally, quality. But consider: making things consistent is about making things conform to a norm. This is why Damian and Petre argued that methodology can be a driver of mediocrity:

“If we assume that practitioners are competent, then what drives their decisions? What do they take from methodology – when do they adopt it, and when do they decline it?

Methodology affords potentially valuable leverage:

- structure
- coordination (standardisation, consistency)
- re-use
- communication (common language)
- sharing artefacts (especially in a potentially diverse context)

The importance of a specified methodology may be greater for less-experienced developers or for organisations that haven’t already evolved their own mechanisms for these things.

There are times when developers interpret methodologies strictly:

- When they are first learning them.
- When they fit their context well.
- When the perceived or experienced benefits outweigh the costs.

There are also times when developers deviate from strict interpretation:

- To adapt to local needs.
- When the cost of adherence exceeds the perceived benefit.
- When the methodology (or its underpinning philosophy) is at odds with an effective existing culture.
- When adherence is too constraining.” [Petre & Damian, 2014]

A useful analogy likens the conventional view of software development methodology to driving a juggernaut down a highway. Methodology suppresses variation, because there is some value to be realised from systematic constraint, consistency, convention, and standardization. It is a structured process conceived in terms of driving toward a specified solution. However, when methodology is embedded in a culture of strict adherence, it takes on a momentum of its own. Methodology is a support, not a replacement, for critical thinking. And one of the things that distinguishes expert designers is just that: persistent and effective critical design thinking.

So how do expert software designers manage the complexity? Instead of trying to manage complexity through a fixed methodology, experts manage it by recognizing that different perspectives must be maintained simultaneously. They use different methods and notations as lenses, changing them deliberately, in order to address different perspectives on the design [Petre and Green, 1990]. We characterise this management of perspectives in terms of design dialogues. These dialogues intersect, contrast, and represent different, simultaneous perspectives on design – and accommodate shifts in the design space as the design evolves. The dialogues may be within the designer’s mind, between a designer and some external representation, or among colleagues. Doing so both clarifies why design is complex and identifies key core considerations that designers keep in mind.

This characterisation is not wholly novel; it draws on and integrates ways that others have characterised design. It is, however, grounded in empirical studies of expert software designers and high-performing teams which are presented elsewhere (an annotated bibliography is available at: <https://softwaredesigndecoded.wordpress.com/annotated-bibliography/>).

The following sections introduce each of the dialogues in turn.

2. Problem - solution

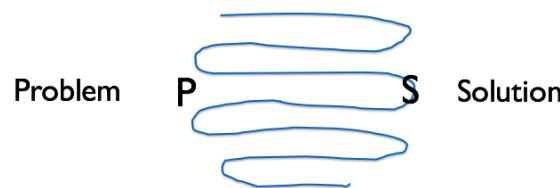


Figure 1: The dialogue between problem and solution

In practice, software designers are presented with a “problem” – perhaps a design brief or presentation or simple verbal cue or a problem ticket in an issue tracker – and are expected to generate a ‘design’, the plan for change in the world that can be realized to make that change in the world happen (the ‘solution’). That ‘design’ may be realized as a schema, design document, UML diagram, mock-ups, code, etc. – depending on the context.

However, design is typically not a straightforward transition from problem to solution; rather, there is a dialogue that takes place between the design problem and the design solution. Dorst and Cross [2001] articulated this *co-evolution of problem and solution* clearly, and Michael Jackson gave prominence to this dialogue in his work, (particularly with *Problem Frames* [2001]), which stressed the importance of understanding the problem in context and in depth, and identifying and decomposing the requirements, in order to map them to a solution.

The design process is about managing the interplay between the problem and solution, as a ‘dialogue’ that emerges. This dialogue between problem and solution clearly includes both problem discovery and understanding, and generating a solution, and each of those includes multiple tasks: information-gathering, sense-making, synthesis.

Why is this dialogue necessary? Well, one could try to understand a design problem *ad infinitum* in the abstract. But ‘the rubber tends to hit the road’ [Anderson, 1998] when designers consider solution and problem in dialogue: new considerations emerge; gaps in the understanding of the problem emerge; the problem may be re-imagined when priorities shift with deeper understanding. Hence Dorst and Cross’s emphasis on “co-evolution”; as the understanding of the design problem evolves, so does the solution space in which it is solved. The dialogue provides a basis for evaluating an evolving solution and its fitness for purpose.

3. Levels of abstraction / focus

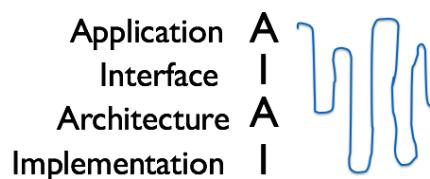


Figure 2: The dialogue between levels of abstraction, that is, between application, interaction, architecture, implementation.

A second dialogue takes place across different levels of abstraction and focus. Design is decision making, involving many decisions of different kinds. Clearly, not all of these decisions are at the same ‘level’ of focus, and, actually, some of these decisions inform others. For example, if we decide that the functionality of the system is to be fault-tolerant, that should inform the choice of platform we are going to use. If we decide to use MVC (model—view-controller) as the dominant architecture, that is going to influence the data structures and APIs that we build. Etc.

Various characterisations of different levels of abstraction have been offered by other authors. We find it useful to distinguish four high-level categories of design:

Application design: “what is the software to do?”

Interaction design: “how does one use the software to do that?” (i.e., interpreting the functionality in terms of user interaction)

Architecture design: “how does it principally solve the problem?” (i.e., realization of both application and interaction design into an overall software solution)

Implementation design: “what are all the details that help make it solve the problem?” (i.e., realization of application, interaction, and architecture design into actual code).

The reality is, just as designers engage in a dialogue between problem and solution, they engage in an emerging dialogue among these four levels of design in an evolving solution space.

This emerging dialogue is not necessarily linear, from application to interaction to architecture to implementation. It may shift from application to architecture and implementation, for instance, to see if a certain piece of functionality can technically be implemented. Or it may move from interaction to implementation, coding up a UI independent of what the underlying architectural framework will be. Sometimes, indeed, designers just have to try some things out. The understandings of each level are imperfect, and they co-evolve as designers shift among the perspectives.

As designers continue to make decisions at all levels, decisions at some levels start constraining decisions that can be made at other levels. For example: Ania could not design a report, because the distributed database would be too slow in generating it, because data was distributed and the join was too expensive. A level of inertia emerged from prior decisions. Sometimes, designers are in a position where they can redo, but as more design decisions are made, they will be more constrained by what they have already decided.

Sometimes designers will purposely co-design parts at different levels and actually be working on multiple perspectives at the same time. For example, we often see UI elements next to architecture elements next to a set of functional requirements on a whiteboard, with the discussion rapidly moving among all three of them, often juxtaposing a pair, and then making updates to each.

Sometimes, work at the implementation or architecture level allows designers to realize that new opportunities arise at the application or interaction level (and similarly for other combinations of levels). For example, Fred could be in the midst of coding, and realize that something that he thought could not be built actually could, if he changes the code around, so he returns to the architecture and updates it – and he may subsequently return to the application and ‘raise the bar’ in terms of the reliability that he wants to achieve.

Those two dialogues – problem/solution and levels of abstraction - co-exist.

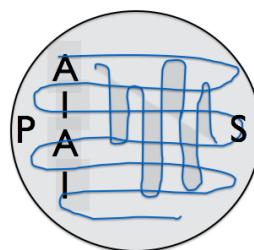
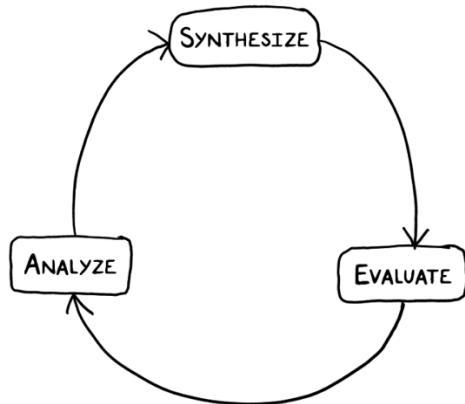


Figure 3: The first two dialogues co-exist and contrast.

4. The design cycle



*Figure 4: The design cycle.
[Figure by Yen Quach; used with permission.]*

Another fundamental concept in design work is the design cycle: analyze -> synthesize -> evaluate [e.g., Lawson, 1997, p. 37]. This can be thought of as an iterative process – or as a set of critical thinking functions or skills – or as another form of dialogue, whose focus shifts in an iterative cycle (and in cycles within cycles), with each focus potentially informing each of the others.

The dialogues between problem and solution, and between the levels of abstraction and focus, occur within iterations of the design cycle dialogue, with the input from one dialogue informing the other.

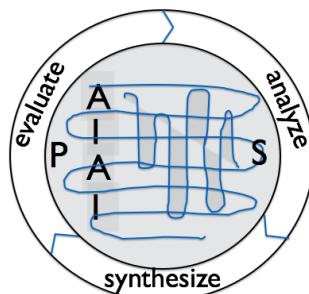


Figure 5: Overlay of the three dialogues so far.

5. Pragmatism and fitness for purpose

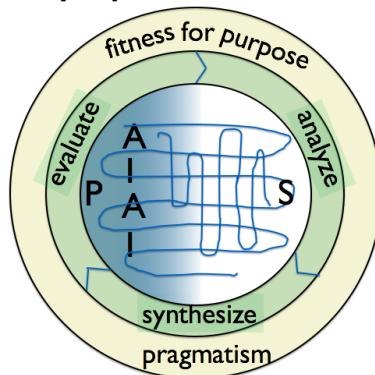


Figure 6: Fitness for purpose / pragmatism

Another crucial conversation takes place: the dialogue of fitness for purpose playing off with pragmatism. It is the dialogue that connects the other two dialogues to ‘reality’ – toward grounded decision making.

Design involves challenges that ultimately lead to the need for satisficing and making trade-offs. We cannot simply make the perfect design solution under given limitations on budget, time and effort. So, we need to engage in a conversation of when ‘good enough is good enough’, which is a conversation that plays out along the issue of desirability versus feasibility. This represents the trade-off between what is ideal from the perspective of the audience and other stakeholders, and what we can actually in the end design and build within the budget, time, effort constraints that we are given. This is where the designer ends up being the pragmatist, the maker of choices of ‘what is in’ and ‘what is out’ and ‘why’.

The fitness-for-purpose/pragmatism dialogue is at the balance point between the well-informed understanding of the problem and the considered choices about what is prioritised in the solution. It takes account of the problem and solution contexts: of the intended purpose, audience, environment-of-use, and of practical constraints. It does not anticipate all possible interpretations and uses; rather it addresses the match between the intended purpose and the feasible solution (and so it relates to the first dialogue – between problem and solution).

6. Dialogues among team members

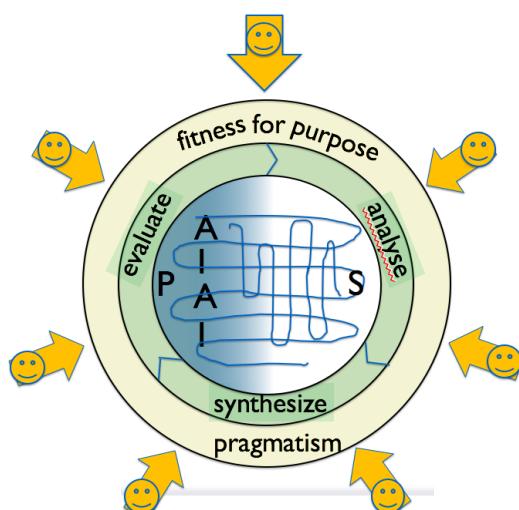


Figure 7: The contrasting design dialogues occur in the context of a design team – and its discussions.

Of course, all of this happens in the context of a design team. The design cycle of analyse - synthesize - evaluate is not only manifest in how one designer thinks in the moment, but it plays an equally important role in how we organize design activities in a broader social design context. It might start at a design meeting at a whiteboard. That conversation may have an overriding design cycle – with smaller design cycles addressing sub-problems or concerns that arise as part of detailed thinking or addressing the concrete ideas and decisions that contribute to progress. That meeting might be part of a sequence of meetings that constitute an over-arching design process. For example, those meetings might be part of an Agile sprint of two weeks. That spring might be part of a larger set of sprints within a given release cycle. And so on.

To sum all of this up: We engage repeatedly the design cycle, and engage in iterations of cycles, and in cycles within cycles. We do this at multiple levels. In doing so, we are engaging in different design dialogues: bridging between problem and solution; considering decisions at four different levels: application, interaction, architecture, and implementation. We integrate these dialogues in terms of the

balancing dialogue - about ‘fitness for purpose’ and pragmatism. This in terms of trade-offs between desirability and feasibility.

It is no surprise, then, that the software development community – organizations, developers, researchers – has tried to impose structure on it. Why? To simplify, so that designers and teams of designers do not have to think about everything at once. And yet, maintaining awareness of all the different factors at play is characteristic of expert designers.

Hence this quotation from Nigel Cross: “Following a reasonably structured process seems to lead to greater design success. However, rigid, over-structured approaches do not appear to be successful. The key seems to flexibility of approach, which comes from a rather sophisticated understanding of process strategy and its control...” [Cross, 2003, p. 116] A structured process helps, but not if it’s rigid, and only with sophisticated understanding, including opportunism and modal shifts.

Cross is indicating what we refer to as the ‘design mindset’ – which combines critical thinking, with an appropriate toolset of methods and analytics, and design thinking, with an openness to opportunity and change that drives innovation. Edward Glaser, in his seminal work on critical thinking and education [1941], defined critical thinking as:

“The ability to think critically ... involves three things: (1) an attitude of being disposed to consider in a thoughtful way the problems and subjects that come within the range of one's experiences, (2) knowledge of the methods of logical inquiry and reasoning, and (3) some skill in applying those methods. Critical thinking calls for a persistent effort to examine any belief or supposed form of knowledge in the light of the evidence that supports it and the further conclusions to which it tends.”

This contrasts – and interacts – well with Nigel Cross’s characterisation of design thinking:

“... the following conclusions can be drawn on the nature of ‘Design with a capital D’:

- The central concern of Design is ‘the conception and realisation of new things’.
- It encompasses the appreciation of ‘material culture’ and the application of ‘the arts of planning, inventing, making and doing’.
- At its core is the ‘language’ of ‘modelling’; it is possible to develop students’ aptitudes in this ‘language’, equivalent to aptitudes in the ‘language’ of the sciences - numeracy - and the ‘language’ of humanities - literacy.
- Design has its own distinct ‘things to know, ways of knowing them, and ways of finding out about them’.” [Cross, 1982, p. 221]

Together, the two capture the both the critical and the creative aspects of the ‘design mindset’.

There is no ‘ideal’ process – although there is a need for structure. The simplifications and idealisations help and may provide focus, but the ‘reality’ of the design process is that it is messy, with lots to consider and juggle. The multiple contrasting dialogues (with associated methods and notations) allow expert designers to manage focus while maintaining awareness.

7. Summary – and invitation to discuss

There is an inherent tension and interaction between these dialogues, which emphasise different views on a (changing) design space. This is a mechanism by which effective designers manage the complexity: each dialogue provides a focus (if not a simplification) for design reasoning, but effective design maintains the interaction between dialogues and makes use of the contrasts between them to achieve design insight.

This characterisation helps to explain both why existing software engineering methodology does not always work and emphasises the importance of an effective ‘design mindset’ – actively critical, with openness to change and opportunity. In expert practice, that mindset – and the dialogues – are supported by socially embedded and reinforced practices that help promote creativity and mitigate bias [van der Hoek and Petre, 2016].

This paper offers this characterisation as a proposition that is grounded in literature in the more general field of design research and in evidence from studies of expert software designers and high-performing teams (although we don't present the evidence here). Our purpose is to open a discussion about the proposition, and its implications for developing software design tools, practice, and education.

References:

- Anderson, B. (1998) Where the Rubber Hits the Road: Notes on the Deployment Problem In Workplace Studies. Xerox PARC Technical Report EPC-1998-108, later published in: Paul Luff, Jon Hindmarsh and Christian Heath (eds.), *Workplace Studies: Recovering Work Practice and Informing System Design*, Cambridge University Press.
- Cross, N. (2003) *Designerly Ways of Knowing*. Birkhäuser Architecture.
- Cross, N. (1982) Designerly Ways of Knowing. *Design Studies*, 3 (4), October, 221-227
- Dorst, K. and N. (2001). Creativity in the design process: co-evolution of problem–solution. *Design Studies*, 22(5), 425–437.
- Glaser, E.M. (1941) An Experiment in the Development of Critical Thinking, Teacher's College, Columbia University.
- Haigh,T. (2010) Dijkstra's Crisis: The End of Algol and Beginning of Software Engineering, 1968-72,
http://www.tomandmaria.com/tom/Writing/DijkstrasCrisis_LeidenDRAFT.pdf [accessed 13 May 2014]
- Jackson, M. (2000) Problem Frames: Analysing & Structuring Software Development Problems. ACM Press / Addison Wesley Professional, ISBN 978-0201596274.
- Lawson, B. (1997) *How Designers Think: The Design Process Demystified*, Third Edition. Architectural Press
- Petre, M., and Damian, D. (2014) Methodology and culture: drivers of mediocrity in software engineering? *FSE 2014* (Visions and challenges track).
- Petre, M., and Green, T.R.G. (1990) Where to draw the line with text: some claims by logic designers about graphics in notation. In: (D. Diaper et al., Eds.), *Human-Computer Interaction—Interact '90*. IFIP, Elsevier Science Publishers (North-Holland), 463-468.
- van der Hoek, A., and Petre, M. (2016) *Software Design Decoded*. MIT Press.

Undergraduate students' learning approaches and learning to program.

Melanie Coles

Bournemouth University

mcoles@bournemouth.ac.uk

Keith Phalp

Bournemouth University

kphalp@bournemouth.ac.uk

Abstract

This study uses the Revised Two Factor Study Process Questionnaire (R-SPQ-2F) to explore undergraduate students' approaches learning to program. The expectation being that students using deep learning approaches will gain higher programming grades than students who use surface approaches. There is strong evidence to support the hypothesis that deep approaches are related to higher grade outcomes, and surface approaches to lower. There is also strong evidence to support the hypothesis that students who '*hate*' programming do less well than those that do not. There is however, no evidence that previous programming experience has an impact upon the student programming grade.

1. Introduction

Discussions about the difficulties involved in learning to program and the best way to teach programming have been part of the research field for decades, with educators reporting difficulties and failure, and dropout rates being high for programming courses (Bennedsen & Caspersen, 2007; Dijkstra, 1982; Hare, 2013; Jenkins, 2002; Mavaddat, 1976; Robins, Rountree, & Rountree, 2003; Simon et al., 2009; Watson & Li, 2014). Recognition that computer programming appears difficult for a high percentage of students, that many students settle for a pass grade, that students grasp programming principles (if ever) at widely varying times and that a very small percentage of students perform extremely well and demonstrate a keen interest in computing, have been reported since programming teaching began in the 60s (Mavaddat, 1976).

If programming is difficult to learn then a corollary of this is a higher failure rate for programming than for other undergraduate subjects. It is an often cited outcome that learning to program is notoriously difficult (Bornat, Dehnadi et al. 2008; Jenkins 2002, Robins et al 2003) however only a few papers fully explore the suggested higher failure rate and attempt to develop evidence to support this supposition (Bennedsen & Caspersen, 2007; Watson & Li, 2014). The findings of both papers suggest the majority of pass rates are in the range 50-80%, with an average of 67.7%.

Learning to programming involves a range of related, but also contentious elements, all of which need to align should a student hope to do well. Jenkins identified a number of factors that relate specifically to the domain of programming and what makes it difficult to learn rather than to the more commonly explored student aptitude for learning to program. Such factors involve the multiple skills and processes required, the language used to teach the students, the educational novelty of students learning to program, the student interest, the image and the pace of teaching (Jenkins, 2002). Many factors intertwine and have an impact upon the student individually: motivation, previous experience, time spent programming, aptitude for programming and student attendance. Environmental factors such as the teaching style, the programming language used, the assessment mechanism and a range of pedagogical interventions can also influence student performance. This paper focuses on and explores students' motivation and learning approaches when studying programming, and also includes students' emotional response to programming and their previous background.

2. Background

Understanding students' approaches to learning and exploring how such approaches relate to the module outcomes for students is clearly valuable information. If we can understand and impact upon students' motivation, can we influence students' success rates? Many interventions used in the teaching of programming may work because they alter the students' motivation and even that an intervention itself is taking place, may alter the students' motivation.

2.1 Motivation and Learning Approaches

Students' motivation towards their studies seems an obvious factor that could impact upon their outcomes on any module regardless of subject. If a student is motivated to succeed there is more likelihood of them achieving that success. However could motivation play a greater role in students learning to program than it does in other courses? Programming needs persistence and practice, students must be motivated to spend time practicing, even if there is no explicit assignment (Jenkins, 2001). The combination of students' motives to learn and the strategy they use determines their learning approach (Everaert, Opdecam, & Maussen, 2017).

Motivation is an abstract concept that is difficult to measure in any meaningful way (Jenkins, 2001), behaviour can be observed or questions can be asked but the true motivation behind behaviour is never certain. Jenkins results showed that the main motivators for students were firstly aspiration, but closely followed by the desire to learn, both classed as extrinsically motivated, rather than the intrinsic motivation of interest in the subject itself. There was little evidence from any of Jenkins' questions that students were interested in programming, with almost 50% of students only doing programming because it was compulsory – something that he cites (*and is probably backed up by most programming instructors*) as a depressing observation (Jenkins 2001).

Students who are more intrinsically motivated are found to perform better, with higher levels of intrinsic motivation leading to higher programming results (Bergin and Reilly 2005). Students with intrinsic motivation usually undertook to learn programming in their own time, had prior programming experience and displayed higher capabilities. Such students engaged in programming meaningfully, showed persistence in *playing with code* and would apply what they had learnt to real world problems, compared to others who approached their work in a more trial-and-error or impulsive fashion (Carbone, Hurst, Mitchell, & Gunstone, 2009). Bergin and Reilly suggest that extrinsic motivation does not appear to impact upon results, so suggesting that the use of grades, rewards or student comparisons are not useful for motivating students and that educator efforts should focus on improving students intrinsic motivation (Bergin and Reilly 2005).

Carbone et al found that students could experience a change in motivation, they could start off intrinsically motivated but then experience a change so becoming extrinsically motivated and vice-versa. This change in motivation could be triggered by a range of factors including: no reward for extra effort, encountering difficulties they could not resolve, perceived waste of time on tasks, and lack of technical skills. The technical skills were further catalogued into: an inability to identify problems, ineffective tinkering, inability to break programming problem down, lack of problem solving skills, and limited debugging skills. Carbone et al also identified some personal skills that impacted upon students' motivation: poor time management, independence (over reliance on others) and attitude toward programming errors (Carbone et al., 2009). It is interesting how changeable and sensitive motivation appears to be to external factors, such that reward (*in the form of a grade*) could alter a student's motivation (*in both directions*), how undertaking additional effort and perceiving no reward (*again from the marker*) could impact negatively on a student's motivation.

2.2 Emotion

The student's emotional response to learning to program has not received much research attention (Chetty & Van der Westhuizen, 2013) possibly due to the scientific, engineering domain and the stereotypical lack of emotion in these subject area. The stereotype associated with the logical approach, for example Mr Spock from Star Trek, seems to exist in isolation from emotion, yet it is evident from interacting with students learning to program that they experience a range of strong emotions. They "hate programming", they "love programming", they find it "frustrating", "challenging", "rewarding" all of which indicate a strong emotional response. It would seem an obvious corollary that such emotion would have an impact upon the student's motivation and so their programming performance. More successful students appear to have a more positive view of programming (Simon et al., 2009), and whilst this does seem evident the further question maybe - is it the higher grades that promote the liking or the liking that promotes the higher grades?

Simon et al in a survey of 697 students enrolled in seven courses at five institutions found that nearly half (48%) of the 2553 comments received were classified as positive. The two most positive categories listed by students were using the words *fun/cool* or *interesting/rewarding*. Nearly a third of the comments made were negative (32%), with the most often response being *hard/difficult* and *frustrating/stressful* (Simon et al., 2009). Does this third that make negative comments go in some way to explain the high failure rate of programming undergraduates; is there a link between this negative emotional response and a lower grade?

Many of my students say things like programming is "*tough but rewarding*", "*very difficult*", "*too complicated*" and the one I have heard the most often "*I hate programming*". Emotions can profoundly affect students' thoughts, motivation and action, positive emotions such as enjoyment of learning may generally enhance academic motivation. Although negative emotions are not always detrimental, for example task-related anger may trigger motivation to overcome obstacles (Pekrun, Goetz, & Titz, 2002).

2.3 Previous background

One of the most important variables affecting general university performance is past academic results (Alam, Billah, & Alam, 2014). Byrne and Lyons found some significance both in student's mathematics and science results from their Irish Leaving Certificate and their programming examination score, although no such significance was found with English or Foreign Language results (Byrne & Lyons, 2001). The higher grades in both maths and science correlated with students programming scores. Other studies have also found that a maths background correlates with students programming performance (Cantwell-Wilson & Shrock, 2001). So is it that students who have an aptitude for science and maths also have an aptitude for programming or is it that the students who undertook the maths and science (an option) were better prepared to succeed at programming?

What about students' previous exposure to programming, a logical conclusion is that students who could already program would do better than those who had not studied it before. Research does seem to support his suggestion, that experience with programming does benefit students (Hagan & Markham, 2000), but the specific language experienced may be the important factor (de Raadt, Hamilton, Lister, & Tutty, 2005).

3. Hypotheses

Following on from the initial literature review four main hypotheses were developed:

H1: Students with a deep approach to learning will gain higher grades in programming than students with a surface approach

H2: Students with a surface approach to learning will gain lower grades than students with a deep approach

H3: Students who can already program or who have studied a programming before starting university will gain higher grades than students who have not.

H4: Students who have negative emotions towards programming will gain lower grades than students who do not.

4. Methodology

4.1 Instrument Used

The Revised Two Factor Study Process Questionnaire (R-SPQ-2F) was used, this questionnaire is suitable for use to evaluate how students learn or how they approach learning. The revised version of the questionnaire has two main scales Deep Approach (DA) and Surface Approach (SA) with four sub-scales: Deep Motive (DM), Deep Strategy (DS), Surface Motive (SM) and Surface Strategy (SS), shown in the Table 1 (Biggs, Kember, & Leung, 2001; de Raadt et al., 2005). Students adopting a surface approach build their view from facts and details of activities with the aim of reproducing material rather than making theoretical connections, while those adopting a deep learning approach seek to understand the material they are studying.

	Surface	Deep
Motive	fear of failure, emphasis is external, from demands of the assessment	intrinsic interest, emphasis is internal
Strategy	narrow target, rote learn memorises information	maximise meaning relates knowledge

Table 1: From Biggs (2001) and de Raadt (2005)

R-SPQ-2F was used, but rather than the generic form it was modified to apply specifically to learning to programming, thus

1. *I find that at times studying gives me a feeling of deep personal satisfaction*
becomes

1. *I find that at times studying programming gives me a feeling of deep personal satisfaction.*
and

7. *I do not find my course very interesting so I keep my work to the minimum.*
becomes

7. *I do not find my programming unit very interesting so I keep my work to the minimum.*

This was to focus the questionnaire specifically on programming rather than on general strategies. As the strategies used would be expected to differ for different disciplines studied. Additional questions were also added to the questionnaire to explore students' previous experience with programming

I can already program

I have completed a programming course (at school or college)

A further question was added to explore students' general emotional response to programming:

I hate programming.

This question was used as it is the most used by the students themselves.

All questions had a five point Likert Scale response, using alpha characters:

- A — this item is *never* or only *rarely* true of me
- B — this item is *sometimes* true of me
- C — this item is true of me about *half the time*
- D — this item is *frequently* true of me
- E — this item is *always* or *almost always* true of me

4.2 Process

The questionnaires were issued to all students present in lectures and seminars on third week of term, so students had only had three weeks of teaching. There were 293 students on the course, of these

- 36 students did not complete both the coursework and the exam, for a variety of reasons and these were removed from the study
- 121 students completed the questionnaire and both the coursework and the exam
- 136 did not complete the questionnaire, or did not complete it fully (*no signature or not all questions answered*). Some students were present but elected not to complete it, others were not present.

The students all undertook the same module (*unit in our terminology*); Principles of Programming (PoP), which is an introductory programming unit, taught in the first semester of the students' first year, no previous programming knowledge was assumed. The students had a two hour lecture and a two hour lab session each week, for 12 weeks. These lectures covered a foundational programming topic, starting with variables and data manipulation, then selection, loops, file reading and writing and finishing with sorting and searching. For the coursework students had to upload multiple tasks every other week (*four different sets of tasks*), and at the end of the 12 week block there was an exam. The four pieces of coursework together give 50%, with the earlier ones being weighted less (5%, 5%, 20% and 20%) of the overall unit total and the exam gives the other 50%.

All questionnaires were then put away until after the module had finished.

4.3 Threats to validity

Some students either elected to not complete the questionnaires or were not present when the questionnaires were issued and such self-selection may have an impact upon findings. Was there a difference in achievement between the students who completed the questionnaire and those that did not? The analysis can be seen in table 2 below.

	Coursework	Exam	Unit Total
Completed	71.0	71.9	71.4
Did not complete	60.6	61.6	61.1

Table 2: Unit Averages

There was a difference for both coursework and exam scores individually and also obviously for the unit total. This may indicate the difference in attendance vs non-attendance in the unit outcomes for the students, those not attending are already engaging in behaviour that may impact negatively on their grades. Students who were present and elected to not complete the questionnaire may be those less interested in the academic discipline and helping with research, or possibly more concerned about the relationship of completing the questionnaire to their programming marks.

As students are self-reporting what they say their approach to learning is and what it really is may differ. Also the fact that they were completing the research study for one of their unit tutors may impact upon their responses to questions, they may have responded as they thought was ‘best’, even though students were assured the questionnaires would not be looked at until after they had finished the unit.

5. Results

The first exploration of the results was to correlate the response to the overall unit average as can be seen in Table 3 below.

Approach	Correlation	Significance
deep approach	0.3374	.000154
surface approach	-0.3584	.000055
I could already program before starting university	0.1432	.11713
I had completed a programming course before starting university	0.1077	.239664
I hate programming	-0.3269	.000263

Table 3: Correlation of questionnaire answers to unit total

So the deep approach is positively and significantly correlated with the student’s unit total and surface approach is negatively correlated, both of which support the H1 and H2 hypotheses. However what is interesting is that neither the students’ (reported) ability to be able to program or their having previously studied programming had a significant correlation with the unit total. So not supporting the H3 hypothesis. Students’ emotional response, the “*I hate programming*” question, is also negatively correlated with the unit total, so supporting hypothesis H4.

5.1 Further Analysis

A more detailed analysis of the data was explored, examining the relationship between each of the sub-scales and the unit assessment element, i.e. either coursework or exam; this can be seen in Table 4 below.

Approach	Against	Correlation	Significance
Difference between deep - surface	unit total	0.3968	.00001
deep motive	coursework	0.3592	.000052
deep motive	exam	0.3139	.000455
deep motive	unit total	0.3760	.000021
deep strategy	coursework	0.1569	.085679
deep strategy	exam	0.2245	.013303
deep strategy	unit total	0.2165	.017073
surface motive	coursework	-0.2636	.003563
surface motive	exam	-0.3307	.000219
surface motive	unit total	-0.3359	.000173
surface strategy	coursework	-0.2142	.018426
surface strategy	exam	-0.2886	.001359
surface strategy	unit total	-0.2849	.001595

Table 4 : Correlation of sub-elements to unit assessment

As the questions for both the deep and the surfaces approaches could all be scored at either A or E, the difference between the scores was calculated (DA minus SA) to see if the difference would also

correlate to student outcomes. As can be seen in Table 4 above this proved significant, so whilst some students may just have been entering As and Bs almost at random there was evidence that students with a high deep approach score and low surface approach score would have improved outcomes in programme.

Looking at the different groups of questions that make up the deep or surface approaches there are more nuanced results. It is interesting to note that there is strong evidence for a relationship between deep motive and coursework, exam and unit total outcomes, which are all significant. However the correlation between deep strategy and coursework is not statistically significant, and there is weaker significance for deep strategy and both exam and unit total. Also of note is that there is only weak evidence for surface strategy to coursework.

6. Discussion

6.1 Deep Approaches

So clearly having an intrinsic interest in programming improves outcomes for students on an undergraduate programming course. Although it could be that those attending and so filling in the questionnaire were more likely to be those interested in programming. However the deep strategies employed appear to have less of an impact upon the outcomes for students. Deep strategy: the five questions that make up this group are:

- *I find that I have to do enough work on a programming topic so that I can form my own conclusions before I am satisfied.*
- *I find most new programming topics interesting and often spend extra time trying to obtain more information about them.*
- *I test myself on important programming topics until I understand them completely.*
- *I spend a lot of my free time finding out more about interesting programming topics which have been discussed in different classes.*
- *I make a point of looking at most of the suggested readings that go with the programming lectures.*

Is it just that such strategies do not apply entirely to programming as a subject? Whilst the last question, following up on reading, is possibly not high on a programmers list of strategies as the staff, the internet and other students are possibly more likely to be used as a source of support. The other four questions are all stereotypical behaviours associated with the archetypal programmer: writing extra code and *playing with code* until you understand it, writing code for fun in spare time. It is interesting that these results suggest that such behaviour is not important to university outcomes for coursework and exams in programming.

6.2 Surface Strategy

Surface strategy when applied to coursework does not appear to be negatively correlated. The five questions that make up this group are:

- *I only study seriously what's given out in class or in the course outlines.*
- *I learn some things by rote, going over and over them until I know them by heart even if I do not understand them.*
- *I generally restrict my programming study to what is specifically set as I think it is unnecessary to do anything extra.*
- *I believe that lecturers shouldn't expect students to spend significant amounts of time studying programming material everyone knows won't be examined.*
- *I find the best way to pass examinations is to try to remember answers to likely questions.*

Such approaches clearly do not have an impact on the quality of the coursework. The coursework submitted by students does focus on a particular topic being covered that week – so for example of writing loops, potentially the assessment used does not suffer when surface approaches are used? Strategy as applied to such coursework would not particularly suffer from a surface approach as once done the student moves onto the next task, that is they become task focused.

6.3 Previous Experience

Previous experience does not impact on unit outcomes, this suggests that approach to study is more important than previous experiences. Previous experience may also have an impact upon both learning approaches and upon emotion before starting their undergraduate course. Many students have anecdotally reported poor experiences of programming at school or college, therefore previous experience could also negatively impact upon results.

6.4 I hate programming

Possibly it is of no surprise that emotional response is negatively correlated with performance. The questionnaire was distributed early in the unit, therefore the response cannot be related to students' coursework grades as they had not yet been returned or potentially to the perceived difficulty of the unit as students do tend to be comfortable with the concept for the first few weeks of programming. However students who had met programming at school previously and who had struggled with it may already have a negative emotional response to programming that does impact upon their unit grades.

6.5 Relationship to Other Subjects

Whilst the questionnaire was focused on student approaches to programming, and there is evidence that the approaches adopted by students do impact upon their programming grades, the obvious further question is ... do these approaches apply to all subjects?

Approach	Against	Correlation	Significance
deep approach	Application of Programming	0.3182	0.00045
surface approach	Application of Programming	-0.3230	0.00034
deep approach	Networks and Cyber Security	0.1611	<i>0.08157</i>
surface approach	Networks and Cyber Security	-0.3481	0.00011
deep approach	Systems Analysis and Design	0.1107	<i>0.23315</i>
surface approach	Systems Analysis and Design	-0.3053	0.00078
deep approach	Computer Fundamentals	0.1826	0.04781
surface approach	Computer Fundamentals	-0.2875	0.00163
deep approach	Data and Databases	0.1830	0.04734
surface approach	Data and Databases	-0.2392	0.00915

Table 5: Correlation of questionnaire answers to other unit totals

There is still strong evidence for deep approaches correlating positively to the second semester programming unit, there is only weak or no evidence for deep approaches correlating to other subjects. Not necessarily to be unexpected as the questionnaire was specifically focused on programming. However what is interesting is the strong evidence that surface approaches negatively relate to all unit outcomes. This suggests that students who use surface approaches for programming use such strategies for all units, and that this has an impact upon their success at university.

7. Conclusion

So whilst there is strong evidence for H1, H2 and H4 from the analysis of the questionnaires, there is no evidence to support H3. This does lead to further questions:

- Is there a relationship between previous experiences of programming and emotional response?
- How can students' approaches to learning be impacted by the tutors?
- Can different assessment strategies impact upon students' approaches?
- Could explicit discussion of students' approaches help them adopt more useful strategies?

Motivation and the learning approaches used by students do appear to impact upon their success rates on an introductory programming module. Deep approaches have a positive impact specifically on programming grades, there is less evidence for other subjects. Surface approaches have a negative impact upon grades for programming and also extend across other subjects, evidencing that such strategies are to the detriment of student performance.

8. References

- Alam, M. M., Billah, M. A., & Alam, M. S. (2014). *Factors Affecting Academic Performance of Undergraduate Students at International Islamic University Chittagong (IIUC)*, Bangladesh. Journal of Education and Practice, Vol.5(No.39).
- Bennedsen, J., & Caspersen, M. E. (2007). *Failure Rates in Introductory Programming*. The SIGCSE Bulletin, Volume 39(Number 2).
- Bergin, S. and R. Reilly (2005). *The influence of motivation and comfort-level on learning to program*. Psychology of Programming Interest Group (PPIG 17). Sussex University.
- Biggs, J., Kember, D., & Leung, D.Y.P. (2001). *The Revised Two Factor Study Process Questionnaire: R-SPQ-2F*. British Journal of Educational Psychology, 71, 133-149.
- Byrne, P., & Lyons, G. (2001). *The Effect of Student Attributes on Success in Programming*. Paper presented at the ITiCSE, Canterbury, UK.
- Cantwell-Wilson , B., & Shrock, S. (2001). *Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors* Paper presented at the ACM SIGCSE 2001, Charlotte, NC, USA.
- Carbone, A., Hurst, J., Mitchell, I., & Gunstone, D. (2009). *An Exploration of Internal Factors Influencing Student Learning of Programming*. Paper presented at the Australasian Computing Education Conference (ACE 2009), Wellington, New Zealand.
- Chetty, J., & Van der Westhuizen, D. (2013). *"I hate programming" and Other Oscillating Emotions Experienced by Novice Students Learning Computer Programming*. Paper presented at the EdMedia: World Conference on Educational Media and Technology, Victoria, Canada.
- de Raadt, M., Hamilton, M., Lister, R., & Tutty, J. (2005). *Approaches to learning in computer programming students and their effect on success*. Paper presented at the Higher education in a changing world Research and development in higher education, Sydney, Australia.
- Dijkstra, E. W. (1982). *How do we tell truths that might hurt?* ACM SIGPLAN Notices, 17(5), 13-15. doi:10.1145/947923.947924
- Everaert, P., Opdecam, E., & Maussen, S. (2017). *The relationship between motivation, learning approaches, academic performance and time spent*. Accounting Education. doi:10.1080/09639284.2016.1274911

- Hagan, D., & Markham, S. (2000). *Does It Help to Have Some Programming Experience Before Beginning a Computing Degree Program?* Paper presented at the ITiCSE 2000, Helsinki, Finland
- Hare, B. K. (2013). *Classroom Interventions To Reduce Failure & Withdrawal In Cs1 – A Field Report* Journal of Computing Sciences in Colleges Volume 28(Issue 5).
- Hawi, N. (2010). *Causal attributions of success and failure made by undergraduate students in an introductory-level computer programming course.* Computers & Education, Volume 54.
- Jenkins, T. (2001). *The Motivation of Students Programming.* Paper presented at the ITICSE, Canterbury, UK.
- Jenkins, T. (2002). *On the Difficulty of Learning to Program.* Paper presented at the LTSN-ICS Conference, Loughborough University.
- Mavaddat, F. (1976). *An experiment in teaching programming languages.* ACM SIGCSE Bulletin, 8(2), 45-59. doi:10.1145/382220.382470
- Pekrun, R., Goetz, T., & Titz, W. (2002). *Academic Emotions in Students' Self-Regulated Learning and Achievement: A Program of Qualitative and Quantitative Research.* Educational Psychologist.
- Robins, A., Rountree, J., & Rountree, N. (2003). *Learning and Teaching Programming: A Review and Discussion.* Computer Science Education, Vol 13(No 2.), p 137-172.
- Simon, B., Hanks, B., McCauley, R., Morrison, B., Murphy, L., & Zander, C. (2009). *For me, programming is ...* Paper presented at the ICER '09, Berkeley, California.
- Watson, C., & Li, F.W.B. (2014, 06/21/2014). *Failure rates in introductory programming revisited.* Paper presented at the Proceedings of the 2014 conference on Innovation & technology in computer science education.

Parlez-vous Java? Bonjour La Monde != Hello World: Barriers to Programming Language Acquisition for Non-Native English Speakers

Brett A. Becker

School of Computer Science

University College Dublin

brett.becker@ucd.ie

Abstract

Learning computer programming could and should be made easier. It is widely accepted that learning to program is fraught with challenges and the literature is not short of work that supports this view. There are many studies related to programming difficulties, barriers, and misconceptions as well as topics such as what language is best for learning and what techniques for teaching programming are most effective. It is often overlooked that globally, the majority of programming students are non-native English speakers. In addition to the barriers faced by all programming students, these non-native English speakers face a substantial class of additional barriers. This is because English is often the language upon which programming languages and their documentation are based, as well as the language of instruction and other environmental conditions.

There have been relatively few studies on the impact of human language on learning programming and the potential barriers this may cause. These barriers also span a wider range than may be obvious upon initial inspection. To complicate matters, natural language issues can add an additional layer of complexity to more universal barriers to learning. For instance it is well known that programming error messages present most novice programmers with difficulty. When these messages are in English as they most often are, any difficulties interpreting them and using them to produce error-free code are most likely compounded for non-native English speakers.

Particularly in a time when broadening participation in computing is a primary objective, the community can no longer afford to overlook the unique barriers faced by non-native English speakers who want to learn to program. This paper discusses these barriers, presents some questions to guide future research, and outlines the author's work-in-progress in the area.

1. Introduction

There have been few studies on the impact of human language on learning programming (Guo, 2018) and the challenges faced by non-native English speakers when learning how to program are poorly represented in the literature (Becker, 2015). This does not mean that it is not a very important area for research. It is likely that there are far fewer programmers whose native language is English than those who are non-native English speakers based on the fact that 95% of the world's population does not have English as their first language (Guo, 2018). However almost all programming languages are designed using English (Veerasamy & Shillabeer, 2014) and most likely the majority of resources and documentation are as well (Guo, 2018; Li & Prasad, 2005). These points alone provide significant justification to studying the differences between how native and non-native English speakers program, and learn to program.

It is also important to note that the treatment of programming languages as similar to natural languages is being discussed and acted upon by many, possibly more outside academic and educational communities than within. It seems that many believe that learning programming is more important than learning foreign natural languages and importantly there is also a reported difference in the degree to which people of different genders believe this.¹ There are also several political movements underway in the

¹<https://www.teachingpersonnel.com/news/people-would-rather-learn-coding-than-a-foreign-language--62462135356>

United States where programming languages may be categorised as a foreign language in curricula² and counted as foreign languages for college entrance requirements³. Less than two years ago Tim Cook remarked “If I were a French student and I were 10 years old, I think it would be more important to learn coding than English. I’m not telling people not to learn English – but this is a language that you can [use to] express yourself to 7 billion people in the world.”⁴ When global technology leaders talk people listen, and there is a serious issue with this message – it implies that English language ability and learning programming are not intricately related. This view neglects to address the fact that there is substantial evidence – some of it bordering on common sense – that those who don’t speak English can be at a real disadvantage when it comes to learning programming compared to native English speakers.

It is not a goal of this paper to provide a comprehensive view of the work on how non-native English speakers learn to program. It is a goal of this paper to set out a discussion on some of the barriers that these students face to inform future work on overcoming these barriers. We also pose some questions that may guide future research on the relationship between programming and natural languages and on the barriers that non-native English speakers may face when learning to program. Finally we present some early work-in-progress in the area.

2. Natural Languages and Programming Languages

Larry Wall, the developer of Perl, whose has a background in linguistics stated that “there is a scale of how much a computer language resembles human language primarily based on how much context is involved”.⁵ Programming languages are not natural languages, however they are languages (albeit artificial, and most commonly written only) that are designed to convey instructions to a computer and therefore have a restricted vocabulary and tightly-defined specifications (Eastman, 1982). However a computer program can (and arguably should) also convey meaning to other humans (Tenenberg & Kolikant, 2014). Further, it would be surprising if programming languages designed by humans did not share characteristics of the natural languages used by the language designers (Naur, 1975).

The relationship between programming and natural languages is complex. It is accepted that parsing natural language by computational means is more difficult than parsing programming languages by the same means. This may indicate that the human parsing mechanism works by other means making it less suitable for parsing programming languages. This would not be surprising, as programming languages were designed to be easily parsed by computational means and natural language evolved along with human brains for millennia. Nonetheless the relationship between natural and artificial languages (and specifically programming languages) is not frequently studied but some work has been done. For instance, Tenenberg and Kolikant (2014) presented several views that relied on multiple established theoretical perspectives on social cognition and human communication, speculating that these may be crucial to understanding how people learn to program computers. Specifically, by casting computer programs as speech acts, they considered that novices learning to program might, can, and sometimes do rely upon their prior and often extensive experience as skilled natural language users. Along similar lines Eastman (1982) demonstrated that programming keywords can be formed using mechanisms analogous to those observed in English such as neologism formation. Miller and Settle (2019) also presented a relationship between natural language and programming in metonymy. We discuss these findings further in Section 4.1.

It is beyond doubt that programming languages and natural languages are related. The extent to which this is true is beyond the scope of this paper. However, even if weak, if this relationship exists to any extent, it is likely that one’s native language affects how a programming language is learned. Regardless

²<https://www.usnews.com/news/stem-solutions/articles/2016-10-13/spanish-french-python-some-say-computer-coding-is-a-foreign-language>

³<https://www.fastcompany.com/3042122/washington-bill-would-count-programming-as-a-foreign-language-on-college-apps>

⁴<https://qz.com/1099791/apples-tim-cook-says-coding-is-better-than-learning-english-as-a-second-language/>

⁵<https://bigthink.com/videos/why-perl-is-like-a-human-language>

of the parallels one draws between programming and natural languages, it is accepted that programmers have to speak ‘computerish’ – we are able to ‘speak’ C, Pascal, SQL or even machine code – and it has been stated that humans learn a computer language using the same faculties as learning natural languages, in an intuitive manner, yet without a profound understanding of what is going on in our brains during this process.⁶ There is also some fresh empirical evidence in this department when it comes to programming languages that supports this hypothesis using human brain studies. In Section 3 we discuss an fMRI study that has provided evidence that code comprehension stimulates the same areas of the brain that natural languages do.

Unlike natural languages which can be quite forgiving due to their ambiguity (and the human ability to interpret that), modern high-level programming languages have a well-defined structure and syntax. Deviating from these specifications renders a program of little use. Therefore it is reasonable to hypothesise that non-native English speakers may be at more of a disadvantage compared to those fluent in English when it comes to code construction, code reading, and debugging. It is also possible that some of these means of interacting with programs may be more severely hampered than others, which requires that these factors be studied on a case-by-case basis. What is clear is that we are operating with high-level languages that are (hopefully) natural-language-like enough for people to use them freely without the need to spend large amounts of time just to figure out what the code should look like and at the same time exact enough for computers to parse it unambiguously.⁶

Another debate that we will not explore here but should be pointed out is ‘teaching’ or ‘pedagogical’ languages (Crestani & Sperber, 2010) vs. ‘real’ languages, which normally refers to languages that are used in industry. Interestingly, teaching languages may have parallels in natural languages when one considers Esperanto. Esperanto is an artificial natural language which has some features of a natural language – just as pedagogical programming languages have some features of industrial programming languages. Interestingly Esperanto even has some native (or first) speakers (Lindstedt, 2006). It should also be noted that there is at least one programming language that is a subset of a natural language. That language is – quite unsurprisingly – English. Inform 7 is a (highly domain-specific) programming language for creating interactive fiction using a natural language syntax. Inform 7 draws on ideas from linguistics and literate programming and is used in literary writing, games development and education.⁷

It is fairly well-known that fluent speakers of multiple natural languages can ‘pick up’ or acquire additional languages with an ease that seems much greater than that of learning one’s first non-native language. Many programmers would say the same for programming languages. Portnoff (2018) makes a case that acquiring a second or subsequent programming language is even easier than it is for natural languages as “they all implement the same set of control and data mechanisms in very similar ways, the task of learning a second programming language for those with in-depth knowledge of a first programming language is more like learning a dialect than an entirely new language” (2018, p. 39). Arguably this makes learning one’s first programming language as easily as possible extremely important as it can be seen as the main key to acquiring other languages, a trait common amongst, and very advantageous for, professional software developers.

2.1. English and Programming Languages

In general non-native English speakers program and learn in English (in as much as one can program in English), as almost all programming languages are designed using the English language as a base (Veerasamy & Shillabeer, 2014). Additionally, most sources of documentation are in English (Li & Prasad, 2005) as are most secondary sources of information such as Stack Overflow. Attempts to develop programming languages using natural languages other than English have been few, and have not gained popularity or use at university level teaching (Veerasamy & Shillabeer, 2014). However, many non-native English speakers, despite using languages that have English keywords choose to use their

⁶<http://www.ppig.org/news/2006-06-01/linguistics-and-programming-languages>

⁷<http://inform7.com/about/>

native language for comments, variable, method and function names.⁸

It is (probably) unlikely that a programming language will ever be created that is equivalent to a natural human language such as English, but being able to construct a computer program with a natural language would be obviously advantageous. Natural language programming, where a high-level programming language is either bypassed or constructed automatically from the input of natural language expressions has been researched for many years but is not currently near a state of useful widespread reality. For a review of such systems, see (Pulido-Prieto & Juárez-Martínez, 2017). However, imagine if perfect natural (English) language programming was achievable today. We can take this to be an extreme case along a continuum where at the opposite end programming languages have a syntax that is completely random. It is quite possible that the programmer's knowledge of English, or any other natural languages for that matter, would be of little use and therefore it is possible that non-native and native English speakers would be on an equal footing. Going back to a perfect (again English) natural language programming reality, it is not hard to imagine that if one can't speak English they have no chance whatsoever in constructing a program. The current situation of high-level languages designed in large part by English language speakers, with English keywords, and English resources, would put non-native English speakers at a disadvantage, but one between these two extremes. Figure 1 depicts this continuum and the hypothetical but plausible difficulty gap between native and non-native learners.

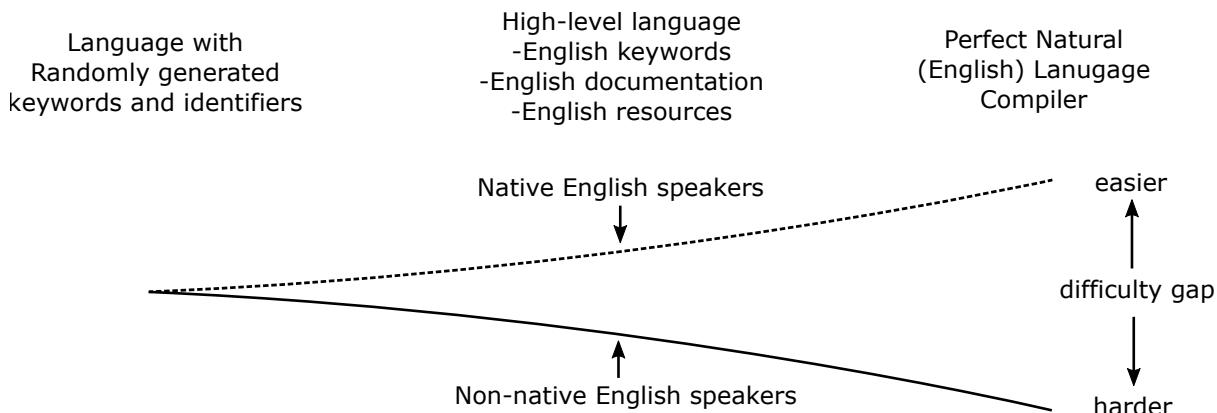


Figure 1 – A hypothetical difficulty gap between native and non-native English speakers grows as one progresses from a language with randomly generated keywords and identifiers through a typical (English) high-level language, through to a perfect (English) natural language compiler

How large this difficulty gap is, and what factors affect it is not well studied. Some influencing factors such as the computer language being used would obviously play a role but we do not currently have definitive answers for seemingly simple questions like: What is more difficult for non-native English speakers to learn, Java or C? It is worth noting that such questions are reasonable because the differences between computer languages, and therefore their differing relationships with natural languages can be substantive; for instance one of the biggest differences between object-oriented and non-object-oriented programming is the possibility to identify the actor of an action using purely syntactic means⁶. There is some recent work that sheds some light on this that we discuss in Section 4.1.

Our lack of knowledge in this area is at least in part because educators are far from agreement on what programming language is best for learning (and for whom), although this debate seems to have gained less attention in recent decades (Becker & Quille, 2019). It is possible that the debate of what language(s) are best for teaching will continue, perhaps indefinitely. However it is worth noting that there is an obvious reason that introductory programming courses rarely use low-level languages like assembly – it is accepted that it is too difficult and perhaps not that useful. Instead we use high-level languages that are by definition more like natural language. This observation alone provides sufficient

⁸<https://softwareengineering.stackexchange.com/questions/1483/do-people-in-non-english-speaking-countries-code-in-english>

```

poiblí aicme DiaDuitDomhan{
    poiblí statach folús príomh(Sreang[] args){
        // Priortáil "Dia duit, Domhan" go dtí an fuinneog teirminéal
        Córás.amach.priontáilln("Dia duit, Domhan");
    }
}

```

Listing 1 – How HelloWorld.java might look with Irish keywords and identifiers

motivation to explore the relationship between programming and natural languages and specifically, how this impacts the way that programming languages are learned.

2.2. An Example

Although to a native English speaker it may seem a somewhat trivial matter to have to deal with non-English keywords and identifiers, consider the fact that in Java the “Hello World” program – traditionally the first program a novice programmer writes – is almost entirely made up of keywords and identifiers. Listing 1 shows what a HelloWorld.java program might look like with Irish language keywords and identifiers, and a comment which is also in Irish.

Unless one reads Irish, it is probably not apparent at all what this program is, or what it would do. It is not unreasonable to assume that to someone who doesn’t know English, or who has a limited grasp of English, would have similar difficulties with the traditional ‘English’ version of the Hello World program.

Anecdotally, the author has had conversations with a Greek colleague who has pointed that ironically, to use L^AT_EX, native Greek speakers have to learn the English names of Greek symbols such as delta (Δ) in order to typeset documents requiring these symbols. Although this may seem trivial, it is a real example of an often overlooked barrier that non-native English speakers can face. It is likely that there are many more examples such as this.

3. Programming Languages: Theories and the Human Brain

It might seem counter-intuitive that the small syntactic footprints of programming languages, with their relatively simple and compact grammars, would translate into a lengthy and involved learning process (Portnoff, 2018). Yet, citing (McCracken et al., 2001; Soloway, Bonar, & Ehrlich, 1983; Tew & Guzdial, 2011) amongst others, Scott and Ghinea (2013, p. 1) concluded that “despite considerable research into programming instruction since the inception of Computer Science as an academic discipline, many learners have not acquired the desired level of competency”.

Evidence on the lack of theoretical approaches to teaching computer programming can be found in a recent review of 5,056 introductory programming papers from the period 2003-2017 – which eventually cited 735 papers – stating: “there are relatively few papers on theories of learning, with no obvious trends across the period of our review” (Luxton-Reilly et al., 2018, p. 66). This paper also provided supporting evidence that many papers on teaching programming don’t have a theoretical foundation. Only 19 of the papers examined were coded as theory-related and the majority of these dealt with “learning styles” which have been largely discredited as “pseudoscience, myths, and outright lies” (Kirschner, 2017, p. 171).

Portnoff (2018, p. 38) also supported this view, stating:

CS educators, however, currently operate with no evidence-based cognitive model for how students learn to program. When partial models have been invoked, they have generally presupposed the involvement of psychological constructs – such as that “cognitive loads” are lowered with drag-and-drop programming interfaces like Scratch or Alice – without having done research (i.e., taking experimental measurements) to corroborate such assumptions.

These are profound observations and insights when one considers the fact that learning programming necessarily involves learning a new (written, artificial) language, and the process of teaching and learning new natural languages has been well studied. Even ‘language-agnostic’ introductory programming courses that use pseudocode in essence require the learner to acquire a new language, and pseudocode has been argued as an unsuitable choice for assessment (Cutts, Connor, Michaelson, & Donaldson, 2014).

Bypassing any intermediate theory, a team consisting of a psychologist, a neurobiologist, a linguist, as well as computer scientists and software engineers, went straight to the source of programming learning – the human brain – conducting a controlled study on brain function of 17 participants using functional magnetic resonance imaging (fMRI)⁹ while they were comprehending short source-code snippets which they contrasted with locating syntax errors (Siegmund et al., 2014). They found a clear, distinct activation pattern of five brain regions, which are related to working memory, attention, and language processing. The authors justly note that “Understanding program comprehension is not limited to theory building, but can have real downstream effects in improving education, training, and the design and evaluation of tools and languages for programmers” (2014, p. 378).

Writing about this study, Portnoff (2018, p. 36) reported: “The programmers in the study recruited parts of the brain typically associated with language processing and verbal oriented processing (ventral lateral prefrontal cortex). At least for the simple code snippets presented, programmers could use existing language regions of the brain to understand code without requiring more complex mental models to be constructed and manipulated.”

What this means for how novices learn to program remains to be seen. It should be noted that fMRI studies such as this do have some threats to their validity – see (Siegmund et al., 2014, p. 385). If such anatomical studies prove to be robust it is likely that new theory will need to be developed, and new experiments carried out to test them, in order to inform the practice of teaching programming most effectively for both native and non-native English speakers.

These results do support the work of Portnoff (2018) who as part of a MSc thesis (Portnoff, 2016), as well as in his practice, argues that implicit (natural) language learning strategies are effective for teaching programming languages to novices. Portnoff found that applying foreign (natural) language pedagogies in programming instruction lead to a dramatic reduction in syntax issues with his students (Portnoff, 2018). He argues that the current prescriptive model of programming language instruction is at odds with the implicit way that native, and second (natural) languages are acquired.

4. Barriers to Programming faced by Non-native English Speakers

Guo (2018) pointed out that non-native English speakers face a range of well-known challenges in English-language classrooms in a wide range of disciplines including math, science, engineering, medicine, and the humanities. These ranged from cognitive to affective to social. Guo also pointed out that often these challenges, such as needing to mentally translate concepts into one’s native language – especially in real time while listening to a lecture – increases extraneous cognitive load and decreases comprehension. Even difficulties with formulating verbal questions, and anxiety about a lack of English fluency makes these students less likely to ask clarifying questions. Bouvier et al. (2016) noted that the contextual background of a problem can also impact cognitive load (regardless of the native language of the student) and that computer science has unique characteristics compared to other disciplines, with the consequence that results from other disciplines may not apply to computer science, thus requiring investigation specifically within computer science. It should also be noted that different languages of instruction can hinder conducting and replicating research into these questions, further hampering progress in the area (Zingaro et al., 2018).

⁹fMRI (Functional Magnetic Resonance Imaging) measures brain activity by detecting changes associated with blood flow. This technique relies on the fact that cerebral blood flow and neuronal activation are coupled. When an area of the brain is in use, blood flow to that region also increases.

Given these well-accepted issues faced by non-native English speakers across many disciplines, and within computer science, it is not unrealistic to hypothesise that these learners may face barriers specific to learning to program. To investigate this, Guo (2018) conducted a survey of 840 responses from programmers spanning 86 countries and 74 native languages, identifying several barriers faced by non-native English speakers. Guo found that these programmers faced barriers with: reading instructional materials; technical communication (listening and speaking); reading and writing code; and simultaneously learning English and programming.

These respondents also expressed a desire for instructional materials to use simplified English without culturally-specific slang, more use of visuals and multimedia, more use of code examples that are culturally agnostic, and the incorporation of inline dictionaries. Additionally, some respondents reported that programming actually served as a motivating context for them to learn English better and helped clarify their logical thinking about natural languages, which provides further support for researching these barriers and how to help students overcome them.

Similarly, but perhaps counterintuitively, Li and Prasad (2005) found that native English speakers preferred examples and practice much more than non-native English speakers, and that non-native English speakers preferred lectures more than native English speakers. They found this to be consistent with their observations, but felt that this was possibly more of a cultural issue than a language issue. Further research needs to be carried out to explore these issues.

Supporting the theory that non-native English speakers face more severe barriers than native English speakers when programming, Dasgupta and Hill (2017) found that novice users who code with their programming language keywords and environment localised into their home countries' primary language (German, Italian, Norwegian Bokmål, Portuguese, and Brazilian Portuguese) demonstrated new programming concepts at a faster rate than users from the same countries whose interface was in the default of English. In developing Spoken Java, a semantically identical variant of Java that is easier to say out loud, Begel and Graham (2005) found several differences between how native and non-native English speakers vocally express in code. An example is the use of Prosody (volume, timbre, pitch, and pauses). They found that the semantic use of prosody was limited mostly to native English speakers – many non-native English speakers who speak English typically use the prosody of their native language, in which pauses, in particular, do not hold the same meaning. This affected how spoken Java was interpreted for instance when dealing with brackets and punctuation.

In the following subsections we explore two very different classes of barriers faced by non-native English speakers when learning to program. It should be stressed that these barriers are present for both native and non-native English speakers, but they might affect these groups differently. These two classes are only two of potentially many more. First we look into a core aspect of programming – dealing with keywords and syntax. We then look into other aspects of the code base – error messages and code comments.

4.1. Syntax, Keywords and Reference Errors

Miller and Settle (2019) explained how novice programmer reference errors are consistent with the use of metonymy, a form of figurative expression in human communication where the name of an attribute is substituted for the name of something closely associated with that attribute; for example ‘suit’ being used as a substitute for ‘business executive’. Miller (2016) provided three possible knowledge sources for why novice programmers produce reference-point errors that are consistent with the use of metonymy. Some of these knowledge sources may differ between native and non-native English speakers, implying that these groups of students may experience the complex relationship between natural and programming languages differently. One of these sources involves misconceptions about notional machines which brings up a question on if native and non-native English speakers may form different models of notional machines. Miller and Settle (2019, p. 2) note that “In contrast to the relative ease with which humans comprehend figurative language such as metonymy, it presents difficulties with human-to-machine communication, particularly in the domain of programming”. They also showed that

the presentation of examples can affect the construction of references in student solutions. They suggest that reference-point errors may be the result of well-practiced habits of communication rather than misconceptions of the task or what the computer can do. As the habits of communication between native and non-native English speakers differ to varying extents, it is most likely that these two groups of students will face different difficulties, or difficulties of varying severity, when it comes to constructing and interpreting programs that contain references influenced by this mechanism.

An examination of keywords in high-level programming languages showed that they are also formed using mechanisms analogous to those observed in the English – for instance, the choice of keywords by language designers is similar to neologism formation in English (Eastman, 1982). A neologism is a new word; it may be either a newly created word or an existing word whose meaning has changed (1982). This process is also related to the choice of identifier names by the programmer. Eastman also noted a conspicuous exception; the use of mirror words such as `fi` to close an `if` statement. This might not be as trivial as it sounds. Mirror keywords can evoke strong reactions. Writing about `fi`, Don Knuth stated: “I don’t really like the looks of `fi` at the moment; but it is short, performs a useful function, and connotes finality, so I’m confidently hoping that I’ll get used to it” (1974, p. 266). In the same paper he stated that Alan Perlis “has remarked that `fi` is a perfect example of a cryptic notation that can make programming unnecessarily complicated for beginners” (1974, p. 266). These reactions are based on the fact that those doing the reacting correctly recognised that they are in fact mirror words. A non-native English speaker might miss this. Therefore it is possible that in some cases, native and non-native English speakers may react differently (and possibly strongly) to keywords. If these groups react differently, it would not be surprising if they find learning and using them to be different experiences. If how these groups use a language differ, it is likely that how they would best learn that language would differ also.

Eastman (1982) put forward a good reason why mirror words could be seen as nonsense – one could almost take them as being random. Interestingly, Stefik and Siebert (2013) carried out four experiments on (largely native English speaking) novice programmer accuracy rates using six programming languages: Ruby, Java, Perl, Python, Randomo, and Quorum. Randomo was designed by randomly choosing keywords from the ASCII table. They found that that Perl and Java – languages using a more traditional C-style syntax – did not afford accuracy rates significantly higher than Randomo, a language with randomly generated ‘gibberish’ keywords. However they found that Quorum, Python and Ruby – languages which do deviate from a traditional C-style syntax – did. One of the main conclusions drawn by Stefik and Siebert (2013) was: syntax does matter to novices and accuracy rates vary by language. Given this it is quite probable that the experience of non-native English speakers would also vary according to language. The question is, how would their experience differ? It is interesting that the results for Randomo were not worse than some well-established languages. This could lead to a hypothesis that at least for these languages, the experience of non-native English speakers may be similar to native speakers. Clearly more work needs to be carried out in this area.

Keyword and identifier names also share similarity to natural language words in that they are often compound. Additionally, abbreviations (and acronyms) are not uncommon in both natural language and programming keywords and identifiers. Keywords made up of parts of existing words can be regarded as blends – something between compounds and acronyms. Suffixes and prefixes are also occasionally used. Guo (2018) provided references that Non-native speakers report struggling to decipher the meanings of code identifiers, especially when they are abbreviated; for instance the C function `getch()` stands for “get character”. Liblit, Begel, and Sweetser (2006) found that programmers choose and use names (for programming constructs) in regular, systematic ways that reflect deep cognitive and linguistic influences. Blackwell (2006) found several categories of vocabulary used in Java documentation revealing extremely complex terminology with similarly complex underlying concepts. These findings also indicate that non-native English speakers may face substantial difficulty in navigating code and documentation.

4.2. Programming Messages and Code Comments

A specific facet of the programming experience that can be affected by natural language ability is dealing with programming messages – error, warning, or other messages resulting from errors with code that result in what are commonly called ‘compiler error messages’. These messages have been shown to be a barrier to learning for students, including both native and non-native English speakers (Becker et al., 2018; Ko, Myers, & Aung, 2004). Arguably these messages, in an ‘English’ programming language, should be comprised of English text that is comprehensible to English speakers, and for native English learners should be easy to interpret, allowing for effective error resolution. An effective message therefore, by definition, should be presented in plain English as much as possible. It is simple to conclude that if native English speakers have trouble with these messages, non-native speakers would have at least as much trouble as native speakers, and in most cases, more. Ko et al. (2004) noted that attempts to translate APIs and error messages have faced a lack of adoption since programmers cannot easily search for online help using the localised terms. It should also be noted that some of the difficulties with error messages such as ‘cascading’ error messages (Becker et al., 2018) likely affect native and non-native English speakers similarly. However, other difficulties likely affect these groups differently.

Programming messages are a part of the programmer-facing code base on the output side. They are intended to be read and interpreted by humans. Similarly, another aspect of programming, but on the input side, are code comments. Like error messages, code comments are an essential part of programming, and somewhat differently to writing code itself, are primarily intended to be read and interpreted by other humans. Comments are written largely in natural language, and therefore require a high degree of fluency in the language being used. The author is unaware of any studies that investigate how non-native English speakers programming in ‘English’ programming languages write comments. The most related work found was Stefik and Siebert (2013) who reported that when creating single-line comments, non-programmers rated the English words `note` and `comment` highly. Interestingly, non-programmers rated the traditional C-style single line comment denotation `\ \` approximately the same as `note` and `comment`. Similar to the discussion in Section 4.1, one could hypothesise that using `note` and `comment` would be more disadvantageous for non-native English speakers. Interestingly though, the results for `\ \` could lead to a hypothesis that there are ways of denoting comments that may be similar in usefulness to native and non-native English speakers. Again, it is clear that more work is needed on this front.

5. Questions

In this section we enumerate some of the questions that arise from the topics discussed in this paper and can be used for the basis of future work in the area.

1. How similar are the processes of learning programming languages and natural languages?
 - (a) How is learning a programming language different for native and non-native English speakers?
 - (b) It is obvious that natural languages are much more difficult to parse by computational means than programming languages. This implies that the human parsing mechanism works quite differently to computational parsing. Does that mean that humans (regardless of natural language) experience a ‘natural’ difficulty in parsing programming languages because of biology?
2. Do non-native English speakers have more difficulty programming specific languages compared to native English speakers?
 - (a) If so, for what languages is the experience for non-native English speakers more similar to that of native English speakers?
3. What techniques can be borrowed from natural language acquisition that would improve programming language acquisition?

- (a) Would non-native English speakers benefit from these techniques in the same way as native English speakers?
4. Apart from the programming language itself, how do different mediums of instruction, and different tools such as IDEs impact how non-native English speakers learn to program?
5. How do specific facets of the programming experience such as keywords, syntax, comments and error messages affect non-native English speakers?

6. Future Work

There have been attempts to create programming languages with non-English keywords, but none have been widely adopted, and attempts to translate APIs and error messages have faced a similar lack of adoption (Guo, 2018). There are also non-English programming environments for languages such as Arabic (Al-Salman, 1996), but these have not been adopted widely (Veerasamy & Shillabeer, 2014).

Inspired by Dasgupta and Hill (2017), the author has piloted a study with approximately 120 non-native English speakers using an online IDE. These students are all native Chinese (Mandarin) speakers enrolled in an introductory programming course as part of Computing/Engineering degrees. The next phase involves a study where similar students will be provided the same IDE with both English and Chinese (Mandarin) interfaces. The IDE currently allows programs to be written in C, Java, and Prolog. The planned research questions are:

1. Do non-native English speakers receiving programming instruction in English prefer to use an IDE in their native language when given the choice?
2. What barriers does the IDE present to non-native English speakers who are learning to program?
3. Is there a correlation between performance and IDE language for non-native English speakers?

7. Conclusion

Learning to program is fraught with challenges and there have been numerous studies over several decades exploring the barriers that novices face when learning to program. However, how non-native English speakers learn to program, and how this differs from native English speakers, is an understudied area.

This paper set out several of the high level issues that non-native English speakers may face when learning to program. There is mounting evidence that there are commonalities between how natural and programming languages are learned, but very little has been carried out on how this would affect non-native English speakers.

It is probable that the barriers that non-native English speakers experience when learning to program are different to those that native English speakers face. Many of these barriers may affect both native and non-native English speakers, but could affect non-native English speakers to a greater extent. Most likely, there are barriers to learning programming that are faced by non-native English speakers that native English speakers do not face.

This paper presented an overview of some of the challenges that non-native English speakers face when learning to program. It also presented several unanswered questions that may lead to future research that may help these students overcome such challenges. It also presented the author's planned work in the area of how the language of the programming environment affects non-native English speakers. It should be noted that this is not a comprehensive literature review, but the author has been collecting papers on the relationship between natural and programming languages and on how non-native English speakers learn to program for years and this paper cites about half of that collection. The interested reader is guided to the following as entry points for further reading Guo (2018); Pal (2016); Portnoff (2018, 2016).

Particularly in a time when broadening participation in computing is seen as a primary objective, the community can no longer afford to overlook the unique barriers faced by non-native English speakers who want to learn to program.

8. Acknowledgements

The author would like to thank Dónal and Mairéad Holohan for their help with the ‘Irish Java’ translation.

9. References

- Al-Salman, A. S. (1996). *An arabic programming environment* (Unpublished doctoral dissertation).
- Becker, B. A. (2015). An exploration of the effects of enhanced compiler error messages for computer programming novices. *Thesis* (November). Retrieved from <https://arrow.dit.ie/ltdis/35/> (<https://www.brettbecker.com/publications>)
- Becker, B. A., Murray, C., Tao, T., Song, C., McCartney, R., & Sanders, K. (2018). Fix the first, ignore the rest: Dealing with multiple compiler error messages. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education - SIGCSE ’18* (pp. 634–639). Baltimore, MD, USA: ACM. Retrieved from <http://dl.acm.org/citation.cfm?doid=3159450.3159453> doi: 10.1145/3159450.3159453
- Becker, B. A., & Quille, K. (2019). 50 Years of CS1 at SIGCSE: A review of the evolution of introductory programming education research. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education - SIGCSE ’19* (pp. 338–344). Minneapolis, MN: ACM. Retrieved from <http://dl.acm.org/citation.cfm?doid=3287324.3287432> doi: 10.1145/3287324.3287432
- Begel, A., & Graham, S. (2005). Spoken programs. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC’05)* (pp. 99–106). IEEE. Retrieved from <http://ieeexplore.ieee.org/document/1509493/> doi: 10.1109/VLHCC.2005.58
- Blackwell, A. F. (2006). Metaphors we program by: Space, action and society in Java. In *18th Workshop of the Psychology of Programming Interest Group - PPIG ’06*. Retrieved from <http://www.ppig.org/library/paper/metaphors-we-program-space-action-and-society-java>
- Bouvier, D., Lovellette, E., Matta, J., Alshaigy, B., Becker, B. A., Craig, M., ... Zarb, M. (2016). Novice programmers and the problem description effect. In *Proceedings of the 2016 ITiCSE Working Group Reports* (pp. 103–118). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/3024906.3024912> doi: 10.1145/3024906.3024912
- Crestani, M., & Sperber, M. (2010). Experience report: growing programming languages for beginning students. *ACM Sigplan Notices*, 45, 229–234. Retrieved from <http://dl.acm.org/citation.cfm?id=1863576>
- Cutts, Q., Connor, R., Michaelson, G., & Donaldson, P. (2014). Code or (not code) – Separating formal and natural language in CS education. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education - WiPSCE ’14* (pp. 20–28). New York, New York, USA: ACM Press. Retrieved from <http://dl.acm.org/citation.cfm?doid=2670757.2670780> doi: 10.1145/2670757.2670780
- Dasgupta, S., & Hill, B. M. (2017). Learning to code in localized programming languages. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale - L@S ’17* (pp. 33–39). Cambridge, MA, USA: ACM Press. Retrieved from <http://dl.acm.org/citation.cfm?doid=3051457.3051464> doi: 10.1145/3051457.3051464
- Eastman, C. M. (1982). A comment on English neologisms and programming language keywords. *Communications of the ACM*, 25(12), 938–940. doi: 10.1145/358728.358756
- Guo, P. J. (2018). Non-native English speakers learning computer programming. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI ’18* (pp. 1–14). New York, New York, USA: ACM Press. Retrieved from <http://dl.acm.org/citation.cfm?doid=3173574.3173970> doi: 10.1145/3173574.3173970

- Kirschner, P. A. (2017). Stop propagating the learning styles myth. *Computers and Education*, 106, 166–171. Retrieved from <http://dx.doi.org/10.1016/j.compedu.2016.12.006> doi: 10.1016/j.compedu.2016.12.006
- Knuth, D. E. (1974, dec). Structured programming with go to statements. *ACM Computing Surveys*, 6(4), 261–301. Retrieved from <http://portal.acm.org/citation.cfm?doid=356635.356640> doi: 10.1145/356635.356640
- Ko, A. J., Myers, B. A., & Aung, H. H. (2004). Six learning barriers in end-user programming systems. *Proceedings - 2004 IEEE Symposium on Visual Languages and Human Centric Computing*, 199–206. Retrieved from <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1372321> doi: 10.1109/VLHCC.2004.47
- Li, X., & Prasad, C. (2005). Effectively teaching coding standards in programming. In *Proceedings of the 6th Conference on Information Technology Education - SIGITE '05* (p. 239). New York, New York, USA: ACM Press. Retrieved from <http://portal.acm.org/citation.cfm?doid=1095714.1095770> doi: 10.1145/1095714.1095770
- Liblit, B., Begel, A., & Sweetser, E. (2006). Cognitive perspectives on the role of naming in computer programs. In *18th Workshop of the Psychology of Programming Interest Group - PPIG '06*. Retrieved from <http://www.ppig.org/library/paper/cognitive-perspectives-role-naming-computer-programs>
- Lindstedt, J. (2006). Native Esperanto as a test case for natural language. *SKY Journal of Linguistics*, 19(SUPPL), 47–55. Retrieved from http://www.linguistics.fi/julkaisut/SKY2006_1/1FK60.1.5.LINDSTEDT.pdf
- Luxton-Reilly, A., Sheard, J., Szabo, C., Simon, Albluwi, I., Becker, B. A., ... Scott, M. J. (2018). Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE 2018 Companion* (pp. 55–106). New York, New York, USA: ACM Press. Retrieved from <http://dl.acm.org/citation.cfm?doid=3293881.3295779> doi: 10.1145/3293881.3295779
- McCracken, M., Wilusz, T., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., ... Utting, I. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working Group Reports From ITiCSE on Innovation and Technology in Computer Science Education - ITiCSE-WGR '01* (Vol. 33, p. 125). New York, New York, USA: ACM Press. Retrieved from <http://portal.acm.org/citation.cfm?doid=572133.572137> doi: 10.1145/572133.572137
- Miller, C. S. (2016). Human language and its role in reference-point errors. In *27th Workshop of the Psychology of Programming Interest Group - PPIG '16*. Retrieved from <http://www.ppig.org/library/paper/human-language-and-its-role-reference-point-errors>
- Miller, C. S., & Settle, A. (2019). Learning to get literal: Investigating reference-point difficulties in novice programming. *ACM Transactions on Computing Education*, 19(3), 1–17. Retrieved from <http://dl.acm.org/citation.cfm?doid=3308443.3313291> doi: 10.1145/3313291
- Naur, P. (1975). Programming languages, natural languages, and mathematics. *Communications of the ACM*, 18(12), 676–683. doi: 10.1145/361227.361229
- Pal, Y. (2016). *A framework for scaffolding to teach programming to vernacular medium learners* (Unpublished doctoral dissertation).
- Portnoff, S. R. (2016). The case for using foreign language pedagogies in introductory computer programming instruction. *ProQuest Dissertations and Theses*. Retrieved from <https://search.proquest.com/openview/8f4a5a498c2ba52b27787cc79041b955/1?pq-origsite=gscholar&cbl=18750&diss=y>
- Portnoff, S. R. (2018). The introductory computer programming course is first and foremost a language course. *ACM Inroads*, 9(2), 34–52. Retrieved from <http://dl.acm.org/citation.cfm>

- ?doid=3211407.3152433 doi: 10.1145/3152433
- Pulido-Prieto, O., & Juárez-Martínez, U. (2017). A survey of naturalistic programming technologies. *ACM Computing Surveys*, 50(5), 1–35. doi: 10.1145/3109481
- Scott, M. J., & Ghinea, G. (2013). Educating programmers: A reflection on barriers to deliberate practice. In *Proceedings of the HEA STEM Learning and Teaching Conference*. Birmingham, UK: Hihgher Education Academy. Retrieved from <http://repository.falmouth.ac.uk/1650/>
- Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., ... Brechmann, A. (2014). Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014* (pp. 378–389). New York, New York, USA: ACM Press. Retrieved from <http://dl.acm.org/citation.cfm?doid=2568225.2568252> doi: 10.1145/2568225.2568252
- Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive strategies and looping constructs: an empirical study. *Communications of the ACM*, 26(11), 853–860. doi: 10.1145/182.358436
- Stefik, A., & Siebert, S. (2013). An empirical investigation into programming language syntax. *ACM Transactions on Computing Education*, 13(4), 1–40. Retrieved from <http://dl.acm.org/citation.cfm?doid=2543488.2534973> doi: 10.1145/2534973
- Tenenberg, J., & Kolikant, Y. B. D. (2014). Computer programs, dialogicality, and intentionality. In *Proceedings of the 10th International Computing Education Research Conference - ICER '14* (pp. 99–106). New York, New York, USA: ACM Press. Retrieved from <http://dl.acm.org/citation.cfm?doid=2632320.2632351> doi: 10.1145/2632320.2632351
- Tew, A. E., & Guzdial, M. (2011). The FCS1: A language independent assessment of CS1 knowledge. In *Proceedings of the 42nd ACM Technical Symposium on Computer science Education - SIGCSE '11* (p. 111). New York, New York, USA: ACM Press. Retrieved from <http://portal.acm.org/citation.cfm?doid=1953163.1953200> doi: 10.1145/1953163.1953200
- Veerasamy, A. K., & Shillabeer, A. (2014). Teaching English based programming courses to English language learners/non-native speakers of English. *International Proceedings of Economics Development and Research*, 70(4), 17–22. Retrieved from http://www.ipedr.com/vol70/004-ICEMI2014_H00006.pdf
- Zingaro, D., Craig, M., Porter, L., Becker, B. A., Cao, Y., Conrad, P., ... Thota, N. (2018). Achievement goals in cs1: Replication and extension. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education - SIGCSE '18* (pp. 687–692). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/3159450.3159452> doi: 10.1145/3159450.3159452

Usability of Probabilistic Programming Languages

Alan Blackwell*, Luke Church**, Martin Erwig***, James Geddes****, Andy Gordon*****, Maria Gorinova*****+, Atilim Gunes Baydin*****+, Bradley Gram-Hansen *****+, Tobias Kohn*, Neil Lawrence*****+, Vikash Mansinghka*****+, Brooks Paige****, Tomas Petricek*****+, Diana Robinson*, Advait Sarkar****, Oliver Strickson***+

*University of Cambridge; **Africa's Voices Foundation; ***Oregon State University; ****The Alan Turing Institute; *****Microsoft Research; *****+University of Edinburgh; *****+University of Oxford; *****+Amazon AI Research; *****+MIT; *****+University of Kent

Abstract

This discussion paper presents a conversation between researchers having active interests in the usability of probabilistic programming languages (PPLs), but coming from a wide range of technical and research perspectives. Although PPL development is currently a vigorous and active research field, there has been very little attention to date to basic questions in the psychology of programming. Relevant issues include mental models associated with Bayesian probability, end-user applications of PPLs, the potential for data-first interaction styles, visualisation of model structure and solver behaviour, and many others. We look forward to further discussion with delegates at the PPIG workshop.

Introduction

This discussion has been convened to consider open research questions, priorities and potential design guidelines relevant to the usability of probabilistic programming languages (PPLs). It provides a short introduction, for the PPIG audience, to the conceptual and operational principles that underlie PPLs. It then discusses two alternative perspectives: firstly an applications perspective, in which there might be potential for broader application of PPL methods in the context of end-user programming if the languages were usable by a wider range of people; and secondly an educational perspective, in which we consider whether PPLs might be a valuable tool for teaching principles of probability, or even as an introduction to programming that takes a fundamentally probabilistic rather than deterministic or imperative view of how computation should be conceived. Finally, two sections present case studies following these perspectives - a visualisation approach that may have educational value, and a “furthest-first” approach to applications.

Background and History

Probabilistic programming is a paradigm (generally embedded within conventional languages) in which the program is constructed as a model defined in terms of relationships between random variables (note this is not program synthesis, or programming by example, which are topics of interest at PPIG, but not the subject of this paper). Typical variables might be a sample data set, observable system output, or latent variables. A key distinction in relation to conventional programming languages is that variables do not have a single value, but should be regarded as defining (or sampling from) a probability distribution of likely values. Program execution consists of making inferences over the structure using a variety of methods — for example Markov chain Monte Carlo (MCMC) (Wingate 2011) or variational inference (Blei 2017). From a user perspective, the overall paradigm is declarative (it might be compared to logic

programming, involving far more extensive mathematical functions), although most PPLs are hosted within a functional or imperative language that allows more conventional expression of data transformations, I/O and so on. Note also that expert programmers of declarative languages (notoriously in the case of the Prolog “cut”) must read them imperatively in order to anticipate execution - a point that has been made by several people (remembering Prolog) when they interact with the PPL community.

There exist a number of different approaches to probabilistic programming that are built around a variety of semantics and inference engines. Broadly speaking, we can collapse these languages into two sets, one set being the first-order probabilistic programming languages (FOPPLs) (van de Meent et al. 2018), the other being the set of universal, or higher-order probabilistic programming languages (HOPPLs) (Staton et al. 2016). FOPPLs such as Stan (Gelman, Lee, and Guo 2015), BUGS (Spiegelhalter et al. 1996), Infer.NET (Minka et al. 2013) and LF-PPL (Zhou et al. 2019) directly constrain the set of models that the user can express as programs, so that the inference performed in such programs is more predictable. But, because of this restriction our modelling capabilities are limited, hence the construction of HOPPLs, universal languages that allow users to express any model imaginable in a Turing-complete fashion. Universal languages such as Church (Goodman et al. 2012), Anglican (Wood, van de Meent, and Mansinghka 2014), Pyro (Bingham et al. 2018), TensorFlow Probability (Dillon et al. 2017, Tran et al. 2017) and PyProb (Baydin et al. 2019) provide users with the ability to compose programs that generate any arbitrary model. But, at the cost that it is impractical to guarantee correctness of the inference result.

Research in this field has primarily been driven by the desire for effective tools to enable statistical and machine learning research, and there has been little specialist attention to studying the usability of PPLs, or designing features that enhance usability. There has also been relatively little attention to PPLs in the human-centric computing, software engineering, software visualisation or visual languages communities, with the exception of a small number of experiments conducted by authors of this paper (systems built by Gorinova and Erwig are discussed below).

The Idea of a Probabilistic Programming Language

The diversity of technical approaches just described mean that there is no single conception of what probabilistic programming provides. There is even less consensus on where PPL approaches might take us in future (for example, when used as an implementation platform for experiments with deep generative models). Nevertheless, it is useful to consider why this paradigm offers a distinctive intellectual appeal, in terms of the role of computation within a scientific enquiry. Let’s consider one of the possible styles of probabilistic programming, in which we focus on simplicity, interpretability, and causality.

Our modern understanding of the world started with a revolutionary insight and discovery. While analysing the data of the position of stars and planets in the night skies, astronomer Johannes Kepler found a function that reliably describes the data, and hence models the movements of planets in the solar system. A simple function was able to capture the big data, supported predictions, and led to new scientific insights.

Finding a function that describes a given set of data has since appeared in different shapes and forms. Carl Gauss had a clear model of how the function should look like, but had to deal with imprecision and uncertainty in the data when he developed least-squares linear regression. And Joseph Fourier’s description of data through trigonometric functions builds a basis of today’s signal processing. In contrast to Kepler’s great feat of finding a new model, subsequent methods have mostly focused on adapting a known (or assumed) model to the data.

In recent years, AI research has made significant impact with neural networks - a universal set of functions that can model a wide variety of data. With higher power computing systems, deep and complex networks can describe many data sets with surprising precision. However, there is a catch: in their universality, neural networks provide little insight about the actual underlying models behind the data. We often struggle to understand exactly how a neural network describes a given set of data: the price of universality.

Like Gauss and Fourier, however, we often do have an understanding of what the model behind the data should look like. For linear regression, for instance, we assume a linear relationship in the data set and could thus replace the potentially huge and complex model of a neural network with a much simpler model featuring just a few parameters. All we need is a method to find meaningful values for the parameters in our model, whatever model we choose. This is one of the valuable opportunities offered by probabilistic programming methods.

Probabilistic programming in this sense combines the descriptive power of simple models with sophisticated methods to adapt the parameters in your model to given data. At the core of probabilistic programming you will find a set of "inference algorithms" not unlike the "learning algorithms" you encounter when training a neural network. However, instead of training a universal neural network using data samples, you write a specific model in a probabilistic programming language and infer its parameters through conditioning on data. As with earlier generations of logic programming, a probabilistic program is not run in the classical sense, but instead makes inferences. AI advocates sometimes say, of neural network optimization, that the system is being 'taught', rather than programmed - but to apply this analogy to probabilistic programming neglects the work done by the PPL programmer, and in particular the potential in some languages to implement alternative inference models (a layer of abstraction where the computation is expressed in more conventional imperative form, e.g. PLDI 2018).

How does statistical modelling relate to probability?

Classical linear regression is built on the principle of least squares: on the idea that there is a single pair of parameter values for which the "error" between model and data is minimal. In reality, however, linear regression hardly ever returns the true underlying parameters exactly, although we expect the result to be close to the true values, at least if the linear model is a good fit for the data. Our confidence in the proposed parameter values will then also increase when more data points are captured by the function. On the flipside, if the linear model is a bad fit for the data in the first place, the algorithm will never produce truly meaningful and accurate values.

In the context of probabilistic programming, we do not seek a single value for each parameter, then hope that these values together make our model a good fit to the data. It is much more natural to think in terms of probability distributions. Instead of proposing the single best value for each parameter, the inference engine will much rather tell you how probable a certain value is.

Think of it this way: if your model really fits your data, the inference engine will single out a range of values for your parameters that makes the entire model a very probable candidate for describing the data. If the model does not fit your data, the inference engine will find that no specific set of parameter values really stands out, and that nothing will make your model a particularly probable candidate for the data.

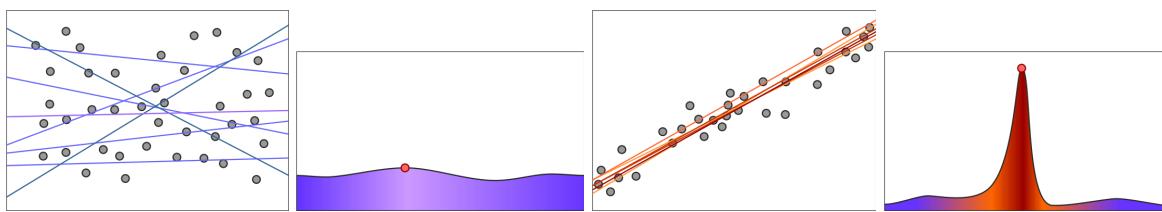


Figure 1: If the linear model is *not* a good fit for the data (on the left), no line really stands out, and all possible parameters have more or less the same probability. Even though we might find a “best” fit, it does not really distinguish itself from the rest. However, if the linear model is a good fit for the data (on the right), then some choices for parameter values are clearly better than others and stand out as highly probable values.

As noted in the introduction to this section, we have presented a relatively straightforward interpretation of probabilistic programming, to illustrate the potential appeal of the paradigm as a scientific tool and educational approach. Current and future developments in PPLs include far more complex approaches to modelling and generation, that might not necessarily retain this intuitive appeal. Nevertheless, this straightforward style of application offers a starting point for broadening access to PPL methods, as discussed in the following sections.

The end-user development perspective on PPLs

It is often observed that more people create programs in spreadsheets such as Excel than in all other programming languages combined (Scaffidi et al 2005). Many business data processing applications that would have required professional programmers to implement them in the 1960s or 70s are now routinely created by people who have never received any formal training in programming, but are able to use spreadsheets to implement a wide variety of straightforward accountancy and data processing applications. The spreadsheet paradigm is approachable in part because of the way that it offers a concrete perspective on the object of interest (the user’s data) rather than on the abstractions of programming. Nevertheless, it is possible to extend the spreadsheet paradigm with sophisticated abstract capabilities such as those of functional programming languages (Peyton Jones et al 2003).

Commercial extensions to the spreadsheet paradigm are generally driven by the business needs of spreadsheet users, who have practical problems to solve rather than being driven by technical curiosity or research agendas. These are defined as end-user programmers (Blackwell 2006, 2017), end-user developers or even end-user software engineers (Ko et al 2011). Increased business interest in the methods of statistical data science suggests that these end-users are likely to find value in PPL capabilities, especially if presented in an interaction context such as a spreadsheet, where users would be able to construct and interpret the behaviour of their program in the context of the data that it relates to. The usability advantages of data-centric presentation have already been observed in previous evolution of statistical applications, such as the adoption of a spreadsheet-style data view in the popular SPSS, when the SPSS-X release under Windows 2.0 included a tabular data editor that became established as the primary user interface for the GUI versions of the product.

Considering PPLs from this data-centric perspective, in terms of the tasks of end-users, raises interesting questions about the boundaries of the programming task. End-users who are creating simple scripts or macros to automate repetitive actions often deal implicitly with the attention investment tradeoff, calculating whether the programming effort will pay off in saved time. Because spreadsheets allow exceptional conditions to be handled by direct manipulation (just changing the cells concerned), spreadsheet programmers are far less likely to devote a lot of effort to anticipating infrequent situations in their code. This is a very familiar situation to data scientists responsible for “data wrangling” - formatting, organising and cleaning data sets for statistical analysis. Although standard research and teaching data

sets have been cleaned in advance, making wrangling distinct from modelling, real-world data science involves a far more ambiguous relationship between the two. It is often difficult to judge whether unexpected data values are errors, outliers, or important clues to an inappropriate model. If probabilistic programming involved closer interaction with original data, this would provide the opportunity for “wrangling” operations to inform the programmed model. It would also provide the opportunity for the mundane tasks of wrangling to be automated through inference, as in the Data Noodles prototype by Gorinova et al. (2016) that allows users to demonstrate how they would like their data to be arranged in a table, then searches for a set of structural transformations that will generate that table.

The same redefinition of boundaries, in a data-centric approach to end-user probabilistic programming, might allow us to revisit the definition of labelling, in the machine learning lifecycle. At present, most supervised learning systems rely on data that has been labelled with a “ground truth” of human interpretation, often obtained via Mechanical Turk, or forced tasks such as ReCAPTCHA. The people who carry out these labelling tasks may have some insight into the modelling assumptions (for example in relation to implicit bias in the judgments they have been asked to make, or explanations of why they made a particular judgment). However, those insights are currently not captured, or even discarded, in conventional machine learning paradigms. There have been some experiments in semi-supervised or mixed-initiative approaches to labelling, for example supporting more dynamic structuring of label categories. However, more sophisticated approaches could be enabled if the data views presented to the labeller offered more direct insight into the structure and behaviour of the model, perhaps even allowing trusted labellers to make incremental adjustments or modifications to the structure. A complementary benefit would be realised by data scientists themselves, who are often advised to spend more time looking at the data, before making assumptions about the structure of the model. Allowing the end-user programmer to contribute to labelling in a way that was continuous with the modelling task allows more sophisticated reasoning across multiple levels of abstraction, in a manner that is analogous to the constant shifts in level of abstraction that are observed in studies of expert programmers (Pennington 1987, 1995)

End-user paradigms such as spreadsheet programming also demonstrate the advantages for learning that result from bridging across levels of abstraction. Modern user interfaces appear more intuitive because a handful of basic principles can be applied in a concrete manner, together with discoverability of more abstract functions and relations. All of these design principles could be applied to implement tools supporting methodologies such as the Bayesian workflow of explore, model, infer, check, repeat (Gabry et al 2019).

We can also consider the potential to generate spreadsheets from PPL specifications. Two of the authors of this report (Geddes and Strickson) are working on a probabilistic programming language -- nocell -- where the result of running a program written in this language is a spreadsheet model applied to the input data. This allows the advantages of spreadsheet models, of understandability and immediacy, to be combined with sophisticated modelling techniques, as well as good software development practices (such as version control and modularity). A further aim is to connect the communities of software-developer data analysts with the wider community of spreadsheet users.

Values in nocell are probability distributions, supporting arithmetic operations and conditioning on observed data. This is motivated in part by the observation that many spreadsheet models are used in situations where capturing uncertainty in the model is beneficial, and that recent advances in PPL and machine language ideas could provide significant value to users of these models, who would otherwise have limited access to tools built on these ideas. This probabilistic approach contrasts with, for example, the type of scenario analysis that is commonly performed, where "typical", "typical-low", "typical-high" and perhaps more extreme values of model inputs are considered, to obtain an idea of the range of values that can be produced by a model (this could still be useful as a /presentational/ tool).

In the nocell approach, the programmer constructs a model as a nocell program, which includes setting appropriate program inputs to probability distributions and perhaps describe observations of their values. When run, this program produces a spreadsheet where the program outputs are evaluated from particular choices of input value, but in addition are annotated with their mean and standard deviation.

An important consideration, driving some of the current work, is how the probability distribution of a value of interest should be represented within the spreadsheet. This should be done in a way that conveys useful information at a level of detail appropriate for a wide audience.

The educational perspective on PPLs

Languages such as Scratch (Resnick et al 2009) include both an application domain (in the case of Scratch, an architecture for agent-based graphical canvas operations) and an educationally-oriented IDE (in the case of Scratch, a block-syntax editor and a library browser). In case of Scratch, one of its major assets is the graphics application domain, echoing the emphasis on graphics in many earlier educational languages from Logo to AgentSheets and Alice. Such languages bring a Piagetian perspective to computation, for example the Scratch sprite or Logo turtle help the learner to think *syntonically* about program execution, by allowing the learner to reason about a computational agent's behaviour (Watt 1998, Pane 2002). Furthermore, as often advocated by Kay, tangible representations can help provide a concrete manipulable representation that helps learners to reason about abstract relations (Repennig 1996, Edge 2006, Kohn 2019).

Beyond the cognitive and notational advantages of a graphical application domain, a core motivation has been that graphics are fun. Children enjoy drawing, and freedom of graphical expression can help bring a creative and exploratory attitude to the introduction of novel notation systems (Stead 2014). How do we make teaching about (Bayesian) probability fun, through use of an application domain that motivates learners? Much traditional teaching of probability follows the traditions of Bayes himself, and other early theorists, in exploring the mathematical implications of gambling (coin tosses, dice throws etc). Teaching of frequentist statistics is largely grounded in the logic of hypothesis testing, and taught in the service of biology or psychology. Might contemporary problems of data science be more motivational as an application domain for education? For example, local children are affected by traffic speed on a road outside their school. The council reports an average speed slightly under the speed limit as evidence that there is no danger. Would children be motivated by gaining access to the council's raw data, and exploring the implications of those distributions for themselves?

A probabilistic programming IDE might potentially include live visualisations of the model; direct dependencies and probability distributions, highlight conditional independencies, as well as provide tools for visual or numerical diagnostics. Some of these ideas have been explored by previous work. For example, Gorinova et al (2016) present a live, multiple-representation environment (MRE) for the probabilistic programming language Infer.NET. Alongside the Infer.NET code, the environment maintains a visualisation of the program as a Bayesian network (a directed acyclic graph, encoding the conditional dependencies between variables). The marginal distribution of each variable is also visualised. Gorinova et al (2016) show that, when presented with debugging and program description tasks, users inexperienced in probabilistic modelling are faster and more confident when using the MRE compared to when using a conventional programming environment. Participants were also more likely to give a higher-level description of the dependencies in the model when using the MRE, as opposed to a lower-level code description. This suggests that live visualisations can be a useful way of teaching core concepts in Bayesian reasoning, and of drawing a clear distinction between conventional and probabilistic programming.

At PPIG 2018, Andrea diSessa made the provocative suggestion that school science lessons such as physics should in future be taught by students constructing their own computational simulations of the phenomenon, rather than through algebraic analysis and fitting of experimental observations. Scientific simulation has been a common application domain for educational programming languages in the past, for example in Repenning's AgentSheets, and Cypher's KidSim. We might imagine the possibility that teaching of probability in schools could be better achieved through modelling in a PPL. Indeed, Goodman and Tenenbaum's *Probabilistic Models of Cognition* includes interactive code examples implemented in WebPPL, Lee and Wagenmakers text on *Bayesian Cognitive Modelling* uses BUGS, and Andrew Gelman's courses at Columbia such as *Statistics GR6103* use modelling in Stan.

Many research users of PPLs have been mathematicians, meaning that the “natural” conceptualisations they are working with may be relatively sophisticated in mathematical terms. In the end-user application field, how far do our priorities shift from support for people who are familiar with the abstract operations, to those who have to treat the models and inferences as black box behaviour? What is the minimum conceptual framework for thinking about the behaviour of Bayesian models?

We can contrast this educational perspective with the suggestion that students should be given a “black box” understanding of how supervised machine learning systems work, as in research by Hitron et al (2019). This project provided students with an experimental environment in which they were able to collect and label (gesture) data, train a classifier, and evaluate the resulting system behaviour. Through experimentation with the system, students did gain improved understanding of supervised learning. We should consider such results in relation to the ICT/computer science debate in school curriculum. The “ICT” perspective was that it was sufficient for students to know how to use applications like Powerpoint or Word, and not necessary to understand how these work internally (i.e. to learn programming). This policy has now been overturned, in favour of teaching at a more fundamental level. On which side of this dichotomy might machine learning fall in future? Will training and using an ML system (for example, predictive text, or spam filtering) be a routine everyday task analogous to the use of Word or Powerpoint, or will it be a sophisticated intellectual task, providing a conceptual foundation for science and engineering? Will students benefit from being able to build new classifiers (using a PPL), or should a standard model be used to describe behaviour, for example in terms of feature selection, convergence, stability and generalisation?

Case study of human-centric design on PPL principles

Here we briefly present an approach to visualize basic computations with probabilistic values, intended to make probabilistic programming accessible to a non-specialist audience, based on a notation introduced in (Erwig and Walkingshaw 2013). The notation is not a comprehensive visualization for PPLs yet; in particular, the visualization of inference requires extensions that we plan to work on in the future. It will also be interesting to compare this approach to probability visualisations such as (Cheng 2011). However, we believe the notation provides a simple metaphor for understanding probabilistic computations and can therefore be the basis for a discussion of a more comprehensive approach to visualizing the execution of PPL programs.

Understanding why and how programs produce their results is important for programmers to be confident in their work and for users to be sure that decisions they make based on a program’s results are valid. Explaining the computation that results from executing a (non-probabilistic) program is a challenging task already, and this task becomes only more difficult for probabilistic programs, since probabilistic values have a more complicated nature and behave differently from deterministic values under transformations.

(The term *probabilistic value* in this context is synonymous with *probability distribution*; it is used to emphasize its role as an object of computational manipulation.)

A general approach to explaining program behavior is based on tracing values or states as they undergo changes when manipulated by a specific program. To apply this approach to the explanation of probabilistic programs we first need a representation of probabilistic values that can be the basis traces. A probabilistic value is a mapping from a set of values to numbers in the range [0, 1]. For example, the result of a fair coin flip can be represented by the mapping {Heads ↦ 0.5, Tails ↦ 0.5}.

Spatial partitions can serve as a simple visual metaphor for illustrating *discrete* probabilistic values (that is, a mapping from a *finite* set of discrete values to numbers in [0,1]). In this representation, we divide a region into blocks, one for each value of the probabilistic value, so that the area occupied by each value corresponds to its probability (Erwig and Walkingshaw 2013). The shape of the area does not matter in principle, but horizontally extended rectangles support the drawing of traces rather well. Here is how the probabilistic coin flip value looks like in this notation.

Coin	
Heads	Tails
50%	50%

We call such a drawing a *probabilistic partition*, or *prop* for short. This notation captures three important features of a probabilistic value, namely (A) the fact that it may consist of multiple values, (B) that each value has a distinctive probability associated with it, and (C) that the probabilities of all values sum up to 1. (The exact position of each value as well as the order among values does not matter.) Note that showing the value probabilities, while helpful to users, is redundant as far as the spatial representation goes, since they are derived from the relative sizes of the partition blocks. Here is another example that represents the result of drawing a ball from an urn that contains the same number of red, green, and blue balls. Of course, the shown percentages are only approximations; depending on user preferences, using a higher precision or showing fractions may be preferable.

Urn		
R	G	B
33%	33%	33%

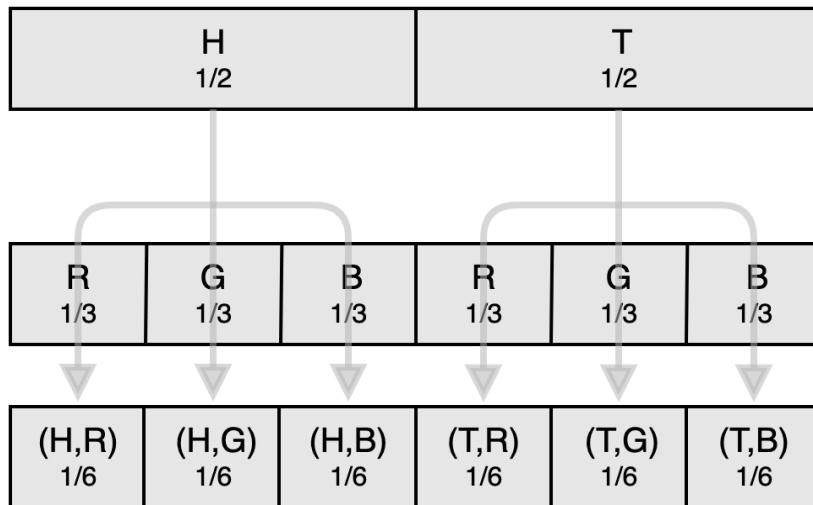
The most basic operation on a probabilistic value is to inquire about the probability of an event, which means to look up the probability in the mapping. In the prop representation this amounts to measuring the size of the occupied area. In general, an event is given by a predicate that selects a subset of values, and the probability for the event is given by the total relative size of the blocks corresponding to the subset of values. For example, the probability of picking either a green or blue ball is represented by $\frac{2}{3}$ of the Urn rectangle.

The programming with probabilistic values involves the transformation of probabilistic values, which raises the question of how to represent (the effect of) operations on probabilistic values. The first basic operation is \cap to compute the joint probability of two events, which amounts to a combination of two probabilistic values such as Coin and Urn into one value Coin \cap Urn. In terms of the employed spatial metaphor, the two props have to be superimposed to create a new prop in which the value combinations have to “share” the common space. For example, if we consider the joint probability of throwing a coin

and drawing a ball from an urn that contains the same number of red, green, and blue balls, we obtain the following prop (abbreviating Heads and Tails).

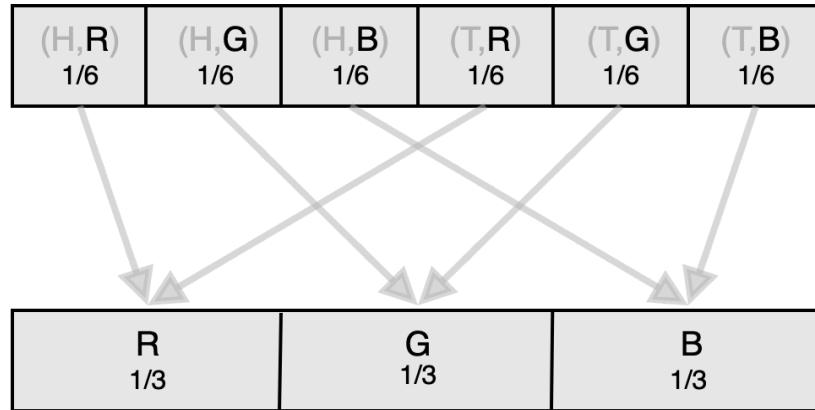
Coin \cap Urn					
(H,R) 1/6	(H,G) 1/6	(H,B) 1/6	(T,R) 1/6	(T,G) 1/6	(T,B) 1/6

The values of the input props are represented as pairs, which reflects the ordering of the arguments of the \cap operation. The generation of the joint probabilities can be illustrated using a flow diagram in which multihead arrows that indicate the flow of values from the first input prop via all the values of the second prop to the resulting prop.



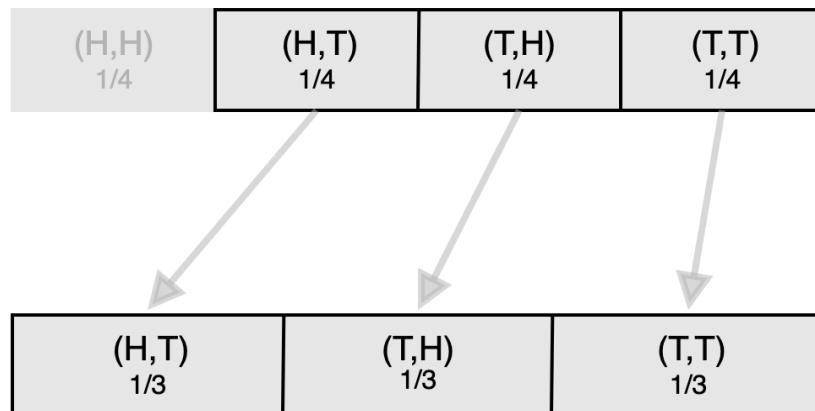
The creation of the result prop can be broken down into a sequence of three basic spatial operations. First, a copy of the second argument prop is created for each value in the first argument prop. Second, each copy is horizontally shrunk to the width of the block of its corresponding value from the first argument prop. Finally, the two partitions are intersected, and the combination of the values and the product of their probabilities annotate the blocks of the resulting partition.

The computation of marginal probability distributions can be also illustrated by a flow diagram. In this case, however, the arrows indicate a spatial union operation of all blocks that have the same value once a value has been marginalized out. For example, we can illustrate the computation of the marginal distribution for Urn as follows.



For each value of Urn we obtain two blocks (one for each value of Coin), and those two blocks are merged into one with a common label. The addition of the probabilities is naturally supported by the spatial metaphor, since the probabilities are homomorphic to the areas.

The computation of conditional probability distributions, which is probably (no pun intended) the most difficult to understand, can be broken down into three steps in the prop representation. We believe that it is this decomposition into simpler operations that provides the explanatory value of the prop representation. As an example, consider the following scenario: While throwing two coins, one comes up Tails. What is the probability that the other one is Heads? Many people believe that the probability is 50%, whereas it is actually 67%. We can illustrate this computation by starting with a joint probability for two coins. The first step is to select the blocks that correspond to the event “one comes up Tails,” which are three blocks except the one containing two Heads. The three blocks define the probability space against which the query “What is the probability that the other one is Heads?” is posed. In the second step, the exclusion of the (H,H) block requires a resizing of the remaining blocks to occupy the whole probability space, which leads to a prop with 3 blocks that each have an associated probability of $\frac{1}{3}$.



The third step requires the grouping of all blocks that match the query “What is the probability that the other one is Heads?” It is easy to see that this event occupies two blocks which occupy $\frac{2}{3}$ of the prop. We can make this step explicit through an additional flow graph that merges the blocks, see (Erwig and Walkingshaw 2013).

At this point we have a method for explaining probabilistic values and computations with them in terms of spatial partitions. Specifically, computing joint probabilities can be explained through the intersection

of partitions, computing marginal probabilities can be explained by relabeling and union of partitions, and computing conditional probabilities can be explained filtering and resizing partitions. Many probabilistic programs are using these basic operations as building blocks, and we can, in principle, apply the partition-based explanation mechanism to explain the execution of such programs. In (Erwig and Walkingshaw 2013) we have used this approach to explain linear programs employing a story-telling metaphor (Erwig 2017), but this approach could also be used to explain non-linear representations as used in probabilistic reasoning in Bayesian networks.

To make this explanation approach practical, more work is required. First, the formalization of props is straightforward by building on our earlier work (Erwig and Schneider 1997). Second, we need a collection of programs as a benchmark for evaluating prop explanations. Third, we probably need several extensions to the notation. For example, to visually explain inference in Bayesian networks we need to represent conditional probability tables. Given the prop notation proposed here, there are some obvious ways for doing it. A more serious challenge is the question of how to scale the notation for bigger programs. This will probably require a notion of partial explanation, see (Cunha et al. 2018) where we also discuss a number of general principles for explanation languages.

A furthest-first initiative: AI tools for African students and researchers

A common strategy in other fields of usability research and human-centric system design is to consider the “furthest first”. This identifies the class of users who are least well served by the current generation of technology or user interface, and gives priority to meeting the needs of those people. It often turns out that a design strategy focused on those who are least well-served results in benefits for all users. Perhaps the most dramatic example of this strategy in programming language research was the Smalltalk language, initially proposed as part of the KiddiKomp project at Xerox PARC (Kay 1996), as one of the first programming languages that would be accessible to children. As it turned out, Smalltalk was more popular among adult programmers than among children (although the underlying principles did continue to benefit very young programmers, in particular through the Smalltalk architecture that underpins the Scratch language). But an even more dramatic outcome of the Smalltalk project was the way in which it required the developers to rethink many other aspects of the programming user interface, leading to the invention of icons, windows, menus, and many other elements of the modern GUI. A furthest-first approach to programming language research can have extraordinarily far-reaching impact.

One of the authors (Blackwell) is currently planning a year-long project, investigating the requirements for probabilistic programming among a population that are currently not well-served by existing languages for AI and data science research. He plans to collaborate with programmers, end-users and students in four different African countries (Uganda, Kenya, Ethiopia and Namibia). This builds on work by Church and others (Church, Simpson et al 2018) designing new tools and architectures for social science, public health and humanitarian research using text data obtained via SMS from regions with poor communications infrastructure. The application of AI methods in such contexts is often intended to empower local actors rather than follow the typical business models of software start-ups, meaning that greater access to configuration and control through accessible programming languages could be particularly important. Economic and political models may also differ from those in typical software technology contexts, for example considering whether those who contribute cognitive labour as a condition of access to media should be paid for their work. The specific aspirations of people in low income countries are also likely to be different, in shaping the imagination of what AI systems can possibly do - providing tools that allow people to explore their own imaginative ideas therefore offers support for innovations that might not have been anticipated in corporate laboratories or universities in wealthy countries.

Some research issues that may be productive in these furthest-first contexts include:

- Redraw system boundaries to consider interaction between labelling and modelling
- Consider reform of school curricula in maths and probability
- Explore AI as enabling structural innovation, not data science as statistical bureaucracy
- Acknowledge the economic and political tensions in cognitive labour such as labelling

We have already noted the specifically educational challenges and opportunities in the design of PPLs. These may present differently in low-income countries, and in relation to the mathematics curriculum taught in those countries. One interesting possibility is the role of probabilistic models in public discourse and activism, for example Carroll and Rosson's investigation of end-user development practices as part of participatory design for community informatics (2007). Experiments such as use of AgentSheets to discuss local community policy (Arias et al 1999) demonstrate the ways that simulation models might be integrated into other social contexts. We might describe this as "broad learning" in contrast to "deep learning", where a wider range of people are able to participate in the definition of models, rather than simply providing training data labels.

If teaching resources are limited, we might also consider following the design strategy of Sonic Pi and other educational languages, in which all tutorial content is integrated into the IDE itself. Sonic Pi has been successfully applied in an African context during the CodeBus Africa project that toured schools in 10 African countries over 100 days in 2017 (Bakić et al 2018).

Conclusion

This discussion represents work in progress, and we expect discussion to continue at the PPIG workshop. Although some initial advances have been reported here (summarising earlier publications), this paper should primarily regarded as a manifesto for promising research directions in the development of more usable PPLs. Several of the authors have substantial research projects in progress, and readers interested in this topic are encouraged to follow developments from those who have contributed.

A key message emerging from our discussions is that the core principles in PPLs are going to be relevant to several very different classes of programmer, and that each of these classes will have very different usability requirements. At present, most users of PPLs are researchers. Researchers do have usability needs, including straightforward considerations of effective software engineering tools (debuggers, tracers, smart editors and so on). It would be possible to carry out more comprehensive task analysis of research work processes, for example as in Marasoiu's study of data scientists, to identify the activity profiles of these researchers and identify ways to optimise tools and notations that suit those profiles. It would also be possible to study the design representations that they already use, and integrate versions of these more closely into data science tools (for example, as in Gordon et al's (2014) Tabular alternative to "plates and gates" diagrams). A second class of programmer is the person who needs to define, explore and apply probabilistic models, but is unlikely to have specialised training (the end-user case). For this class, building on familiar representation conventions such as spreadsheets is likely to be valuable, in addition to supporting more casual and data-centric workflows. A third class is the pedagogic application, where the users are students acquiring an understanding of data science methods or even simple principles of Bayesian probability. For this class, conceptual clarity, minimal distracting syntax, and correspondence to naturalistic descriptions is likely to be important. Many standard considerations of software engineering, including debugging, version control etc have little relevance to the pedagogic situation, beyond the simple need to avoid distraction.

A Probabilistic Postscript

Two of our authors (Advait Sarkar and Tobias Kohn) will not be available to attend the PPIG workshop, because they are getting married on the same day. Not to each other. We invite readers to create a PPL model that would estimate the prior likelihood of such an event, for any given research publication, and thus assess the risk that this might occur again in future. We also record our congratulations to Advait and Tobias!

References

- Arias, E. G., Eden, H., Fischer, G., Gorman, A., & Scharff, E. (1999, December). Beyond access: Informed participation and empowerment. In Proceedings of the 1999 conference on Computer support for collaborative learning (p. 2). International Society of the Learning Sciences.
- Bakić, I., Puukko, O., Hämäläinen, V., and Subra, R. (2018). CodeBus Africa Study. Aalto Global Impact; Aalto University. Washington, DC : World Bank Group.
<http://documents.worldbank.org/curated/en/357931528940784006/Codebus-Africa-Study>
- Baydin, A. G., Pearlmutter, B. A., Radul, A. A., Siskind, J. M. (2018). Automatic Differentiation in Machine Learning: a Survey. *Journal of Machine Learning Research (JMLR)* 18 (153): 1–43.
<http://jmlr.org/papers/v18/17-468.html>.
- Baydin, A. G., Shao, L., Bhimji, W., Heinrich, L., Meadows, L. F., Liu, J., Munk, A., Naderiparizi, S., Gram-Hansen, B., Louppe, G., Ma, M., Zhao, X., Torr, P. H. S., Cranmer, K., Lee, V., Prabhat, Wood, F. (2019). Etalumis: Bringing Probabilistic Programming to Scientific Simulators at Scale. In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC19), November 17–22, 2019.
- Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradham, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., Goodman, N. D. (2018): Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, 2018.
- Blackwell, A.F. (2006). Psychological issues in end-user programming. In H. Lieberman, F. Paterno and V. Wulf (Eds.), End User Development. Dordrecht: Springer, pp. 9-30
- Blackwell, A.F. (2017). End-user developers - what are they like? In F. Paternò and V. Wulf (Eds). New Perspectives in End-User Development. Springer, pp. 121-135.
- Blei, D. M., Kucukelbir, A., McAuliffe, J. D.. (2017). Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877.
- Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., & Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of statistical software*, 76(1).
- Carroll, J. M., & Rosson, M. B. (2007). Participatory design in community informatics. *Design studies*, 28(3), 243-261.
- Carroll, J.M., Rosson, M.B., Isenhour, P., Ganoe, C., Dunlap, D., Fogarty, J., Schafer, W. and Van Metre, C., 2001. Designing our town: MOOsburg. *International Journal of Human-Computer Studies*, 54(5), pp.725-751.

- Cheng, P. C. H. (2011). Probably good diagrams for learning: representational epistemic recodification of probability theory. *Topics in Cognitive Science*, 3(3), 475-498.
- Church, L., Simpson, A., Zagoni, R., Srinivasan, S. and Blackwell, A.F. (2018). Building socio-technical systems for representing citizens voices in humanitarian interventions. In S. Tanimoto, S. Fan, A. Ko and D. Locksa (Eds), *Proceedings of the Workshop on Designing Technologies to Support Human Problem Solving*. University of Washington. pp. 19-21.
- Cunha, J., Dan, M., Erwig, M., Fedorin, D. and Grejuc, A. (2018). Explaining Spreadsheets with Spreadsheets. *ACM SIGPLAN Conf. on Generative Programming: Concepts & Experiences (GPCE'18)*, 161-167.
- Dillon, J. V., Langmore, I., Tran, D., Brevdo, E., Vasudevan, S., Moore, D., Patton, B., Alemi, A., Hoffman, M., Saurous, R. A. (2017): Tensorflow distributions. arXiv preprint arXiv:1711.10604.
- Du Boulay, B. (1986): Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2(1), pp. 57-73.
- Edge, D., & Blackwell, A. (2006). Correlates of the cognitive dimensions for tangible user interface. *Journal of Visual Languages & Computing*, 17(4), 366-394.
- Erwig, M. (2017). *Once Upon an Algorithm - How Stories Explain Computing*. MIT Press, Cambridge, MA.
- Erwig, M. and Walkingshaw, E. (2013). A Visual Language for Explaining Probabilistic Reasoning. *Journal of Visual Languages and Computing*, Vol. 24, No. 2, 88-109.
- Erwig, M. and Schneider, M. (1997). Partition and Conquer. *3rd Int. Conf. on Spatial Information Theory (COSIT'97)*, LNCS 1329, 389-408.
- Gabry, J., Simpson, D., Vehtari, A., Betancourt, M., & Gelman, A. (2019). Visualization in Bayesian workflow. *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, 182(2), 389-402.
- Gelman, A., Lee, D., Guo, J. (2015): Stan: A Probabilistic Programming Language for Bayesian Inference and Optimization. *Journal of Educational and Behavioral Statistics*, 40(5):530-543, 2015.
- N. D. Goodman, J. B. Tenenbaum, and The ProbMods Contributors (2016). Probabilistic Models of Cognition (2nd ed.). Retrieved 2019-6-25 from <https://probmods.org/>
- Goodman, N., Mansinghka, V., Roy, D. M., Bonawitz, K., Tenenbaum, J. B. (2012): Church: a language for generative models. arXiv preprint arXiv:1206.3255.
- Gorinova, M.I., Sarkar, A., Blackwell, A.F. and Syme, D. (2016). A Live, Multiple-Representation Probabilistic Programming Environment for Novices. In *Proceedings of CHI 2016*, pp. 2533-2537.
- Tom Hitron, Yoav Orlev, Iddo Wald, Ariel Shamir, Hadas Erel, and Oren Zuckerman. 2019. Can Children Understand Machine Learning Concepts?: The Effect of Uncovering Black Boxes. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Paper 415, 11 pages. DOI: <https://doi.org/10.1145/3290605.3300645>
- Hoffman, M. D., Gelman, A. (2014). The No-U-turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15(1):1593–1623.
- Gabry, J., Simpson, D., Vehtari, A., Betancourt, M., & Gelman, A. (2019). Visualization in Bayesian workflow. *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, 182(2), 389-402.

- Gordon, A. D., Graepel, T., Rolland, N., Russo, C., Borgstrom, J., & Guiver, J. (2014). Tabular: a schema-driven probabilistic programming language. In ACM SIGPLAN Notices (Vol. 49, No. 1, pp. 321-334). ACM.
- Alan C. Kay. 1996. The early history of Smalltalk. In History of programming languages---II, Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. (Eds.). ACM, New York, NY, USA 511-598. DOI: <https://doi.org/10.1145/234286.1057828>
- Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A.F., Burnett, M., Erwig, M., Lawrence, J., Lieberman, H., Myers, B., Rosson, M.-B., Rothermel, G., Scaffidi, C., Shaw, M., and Wiedenbeck, S. (2011). The State of the Art in End-User Software Engineering. ACM Computing Surveys 43(3), Article 21.
- Kohn, T., Komm, D.: Teaching Programming and Algorithmic Complexity with Tangible Machines. In: Pozdniakov, S., Dagien, V. (eds): Informatics in Schools. Fundamentals of Computer Science and Software Engineering. ISSEP 2018. Lecture Notes in Computer Science, vol. 11169, Springer, Cham.
- Le, T. A., Baydin, A. G., Wood, F. (2017): Inference compilation and universal probabilistic programming. In Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS), volume 54 of Proceedings of Machine Learning Research, pages 1338–1348, Fort Lauderdale, FL, USA, 2017. PMLR.
- Lister, R. (2011). Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. 13th Australasian Computer Education Conference (ACE 2011).
- Minka, T., Winn, J., Guiver, J., Knowles, D. (2013): Infer.net 2.4, Microsoft Research Cambridge. URL: <http://research.microsoft.com/infernet>.
- Pane, J. F., Myers, B. A., & Garlan, D. (2002). A programming system for children that is designed for usability (Doctoral dissertation, School of Computer Science, Carnegie Mellon University). <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.481.2364>
- Pennington, N. (1987). Comprehension strategies in programming. In Empirical studies of programmers: second workshop (pp. 100-113). Ablex Publishing Corp..
- Pennington, N., Lee, A. Y., & Rehder, B. (1995). Cognitive activities and levels of abstraction in procedural and object-oriented design. Human-Computer Interaction, 10(2), 171-226.
- Peyton Jones, S., Blackwell, A and Burnett, M. (2003). A user-centred approach to functions in Excel. In Proceedings International Conference on Functional Programming, pp. 165-176.
- Repenning, A., & Ambach, J. (1996). Tactile programming: A unified manipulation paradigm supporting program comprehension, composition and sharing. In Proceedings 1996 IEEE Symposium on Visual Languages (pp. 102-109). IEEE.
- Resnick, M., Malone, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J.S., Silverman, B. and Kafai, Y.B., (2009). Scratch: Programming for all. Comm. ACM, 52(11), 60-67.
- Scaffidi, C., Shaw, M., & Myers, B. (2005). Estimating the numbers of end users and end user programmers. In 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05) (pp. 207-214). IEEE.
- Spiegelhalter, D., Thomas, A., Best, N., Gilks, W. (1996): BUGS 0.5: Bayesian Inference Using Gibbs Sampling Manual (version ii). MRC Biostatistics Unit, Institute of Public Health, Cambridge, UK, pages 1–59,

- Stan Development Team. (2017). ShinyStan: Interactive visual and numerical diagnostics and posterior analysis for Bayesian models. *R Package Version, 2*.
- Staton, S., Wood, F., Yang, H., Heunen, C., Jammar, O. (2016): Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In 2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pages 1–10. IEEE, 2016.
- Stead, A., & Blackwell, A. F. (2014). Learning syntax as notational expertise when using drawbridge. In Proceedings of the Psychology of Programming Interest Group Annual Conference (PPIG 2014) (pp. 41-52).
- Tran, D., Hoffman, M. D., Saurous, R. A., Brevdo, E., Murphy, K., Blei, D. M. (2017): Deep probabilistic programming. arXiv preprint arXiv:1701.03757
- Van de Meent, J.-W., Paige, B., Yang, H., Wood, F. (2018): An Introduction to Probabilistic Programming. arXiv e-prints, Sep 2018
- Watt, S. (1998). Syntonicity and the psychology of programming. Proceedings of PPIG 1998.
- Wingate, D., Stuhlmüller, A., Goodman, N. (2011). Lightweight implementations of probabilistic programming languages via transformational compilation. In Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, pages 770–778.
- Wood, F., Van de Meent, J.-W., Mansinghka, V. (2014): A new approach to probabilistic programming inference. In Artificial Intelligence and Statistics, pages 1024–1032, 2014.
- Zhou, Y., Gram-Hansen, B. J., Kohn, T., Rainforth, T., Yang, H., Wood, F. (2019): LF-PPPL: A low-level first order probabilistic programming language for non-differentiable models. arXiv preprint arXiv:1903.02482

Towards a Consensus about Computational Thinking Skills: Identifying Agreed Relevant Dimensions

Bostjan Bubnic

Faculty of Electrical Engineering and
Computer Science
University of Maribor
b.bubnic@gmail.com

Tomaz Kosar

Faculty of Electrical Engineering and
Computer Science
University of Maribor
tomaz.kosar@um.si

Abstract

Research on Computational thinking (CT) has already entered its second decade, but still lacks a clear definition that researchers would agree upon. There are even suggestions that the definition of CT is not indispensable and that researchers should focus on other aspects, such as how to include CT in courses, curriculums and how to observe the acquisition of CT. However, it is generally agreed that CT is an important skill within the computer science, while it also extends beyond computing as being a fundamental skill for problem solving in all scientific and engineering disciplines. Moreover, there is a great interest of researchers and educators to explore how to include CT in K-12 context. Our study builds upon the consensus, that multiple skills are involved in CT. Based on the literature review, this study tries to identify a basic, domain independent dimensions of CT that researchers agree upon. The results of this study identify abstraction and algorithms as relevant, domain independent dimensions to build upon consensus. We hope that the study results will encourage further research towards consensus about general, domain independent set of skills that forms CT. This would be particularly beneficial in assessing and teaching CT.

1. Introduction

The phrase computational thinking (CT) was introduced by Seymour Papert in his seminal book *Mindstorms: Children, Computers, and Powerful Ideas* (Papert, 1980, p.182). His main objective for teaching Logo, an educational programming language, was to improve the student's ability to think procedurally. Lately, CT has seen enormous growth in popularity, after it was reintroduced and popularized by Wing in 2006. Until recently, computing has been considered as a domain specific set of problem solving skills possessed by computer scientists, mathematicians and engineers. However, in a seminal article published in 2006, Wing described CT as a way of "solving problems, designing systems, and understanding human behavior by drawing on the concepts fundamental to computer science" (Wing, 2006, p.33). She exposed the ability to think recursively, the use of abstraction and decomposition as important skills when dealing with complex tasks. She also argued that "computational thinking is a fundamental skill for everyone, not just for computer scientists" (Wing, 2006, p.33). Since then, CT has received increasing attention among researchers and practitioners within the field of education. A development of general-purpose problem solving skills across the curriculum is the primary aim of integrating CT into various national curriculums.

As CT originates from computer science and programming, the common approach to learn and assess CT today is by educational programming languages like Scratch, Alice or Snap! It is widely agreed that programming is a useful, but domain specific skill, while CT is applicable to wider educational spectrum and in everyday life. The shift from computing to the general problem solving domain has stimulated researchers to investigate a broader aspects of CT. Recently, CT research in science, technology, engineering and mathematics (STEM) disciplines emerge as prevalent (Khine, 2018). Furthermore, there are several researches exploring CT within non-STEM disciplines, such as language lessons (Sabitzer et al., 2018) and music notation (Barate et al., 2017). Broader CT spectrum presents great opportunities but also challenges for educational researchers, primary and secondary level educators as well as computer science educators. To determine effective methods for teaching, learning and assessing CT, a definition of CT and its scope is needed. Moreover, the main components of CT need to be identified.

To date, there is no consensus concerning the definition of computational thinking. In addition to different perspectives about the definition, there is even a disagreement about the importance of

achieving consensus. On the one hand, researchers are advocating the necessity of CT definition, like Selby and Woppard (2014), Werner et al. (2012) or Dong et al. (2019) who argue that "there is a need for a CT definition that teachers can effectively use to communicate CT to students in core classroom settings" (Dong et al., 2019, p.906). On the other hand, researchers argue that the focus should be on other aspects, such as how to include CT in classes, curriculums and how to observe the acquisition of CT. In this context, Aho argues that "any static definition of computational thinking likely would be obsolete 10 or 20 years from now" (National Research Council 2011, p.37). CT definitions are out of scope of this study.

Apart from CT definition, researchers have begun to characterize computational thinking by means of CT taxonomies and frameworks, where particular concepts and practices were observed. Moreover, it is generally agreed that multiple skills are involved in CT. The main contribution of this paper is the identification of general, domain independent components of CT that will encourage further research towards consensus about general, domain independent components that forms CT. An additional contribution is related to unveiling the characteristics of relevant CT components with an emphasis on computational, non-computational, cognitive and epistemological aspects of particular component.

2. Related Work

It is generally agreed that multiple skills are involved in CT. Wing (2008) addressed the significance of CT concepts as part of "fundamental questions: What are the elemental concepts of computational thinking?" (Wing, 2008, p.3270).

However, there is still no consensus regarding the definitive or necessary components of CT. When computational thinking was reintroduced by Wing (2006), abstraction and decomposition were the fundamental components, while heuristic reasoning, problem reformulation, recursion and systematic testing were referenced as important cognitive processes for solving problems efficiently. The initial set of components were later refined with automation (Wing, 2008). After initial component conceptualization, researchers have begun to characterize computational thinking by means of CT taxonomies and frameworks, where particular concepts and practices were observed. Brennan and Resnick (2012) proposed a programming specific CT framework, where components were associated with programming specific concepts and artefacts. Weintrop et al. (2016) constructed a computational thinking taxonomy for mathematics and science. They "narrow the scope of computational thinking away from generalities, providing a sharper definition that is distinct from computer science". (Weintrop et al., 2016, p.128). Based on literature review, Shute et al. (2017) presented CT components that are "common among researchers".

By examining the literature we found only two papers researching on achieving the consensus of the CT components. Barr et al. (2011) report on consensus that emerged regarding the essential elements of CT during the meeting of a diverse group of educators with an interest in CT from higher education, K-12 and industry. Rich and Langton (2016) used Delphi process to formalize CT definition, where CT components were also part of consensus goals.

2.1. Terminology

When addressing the multiple dimensionality of CT, researchers, educators and practitioners use the terminology interchangeably, not being consistent. Brennan and Resnick (2012) propose a CT framework where components are classified into computational concepts, computational practices and computational perspectives. However, the proposed computational concepts are domain specific, because they address the programming constructs only, such as sequences, loops, conditionals. The computational practices category includes problem solving practices that occur in the process of programming, such as experimenting, iterating, abstracting and modularization. On the contrary, Barr and Stephenson (2011) propose the category of computational thinking concepts that includes abstraction, decomposition and automation, being general and domain independent dimensions. While reviewing literature, we found that authors use different terms when they address CT dimensions, such as skills, practices, components, concepts or abilities and that they are not consistent in their usage. We will be using the term "CT concept" when referring to theoretical, abstract, domain independent dimensions of CT, while the term "CT practice" will be used for domain specific, practical, tangible dimensions. Weintrop et al. (2016) report they moved from term "skill" to a broader

and more actionable term "practice based" on suggestions from many teachers from STEM disciplines.

3. Literature review

Although the purpose of this work was not a systematic literature review, we tried to follow the protocol defined by Kitchenham and Charters (2007) as much as possible.

3.1. Scoping

RQ1: What are the relevant theoretical, domain independent dimensions of CT? RQ2: What is the proportion of intended usage of relevant CT dimensions in the literature?

Answering RQ1 would enable us the examination of the widest possible scope of domain independent CT components. Additionally, the components with the highest number of occurrences could be selected as prime prospects for achieving the agreement about the definitive and necessary components of CT. Answering RQ2 would show the proportion of particular areas of interest and the scope of identified CT components within literature.

3.2. Planning and identification

To identify the broadest scope of CT aspects, we included the following terms as keywords for document screening and snowballing: CT components, CT skills, CT practices, CT dimensions, CT aspects and CT abilities.

The first step in the creation of our initial corpus of relevant papers was the review of the existing CT literature with the aim to identify the broadest scope of CT aspects. Our investigation began by investigating four literature review papers. Inductive qualitative content analysis was conducted by Kalelioglu et al. (2016). They reviewed 125 papers researching on CT to provide a framework about the notion, scope and elements of CT. Hsu et al. (2018) conducted a meta-review of CT studies published in academic journals from 2006 to 2017. Based on the analyses of various educational CT aspects, mostly focusing on K-12 domain, they generated a list of 19 CT components while providing relevant references for each component. Systematic literature review was also conducted by Lockwood and Mooney (2018) to get an overview of the work that has been carried on in the secondary education. Moreover, their objective was to identify potential gaps and opportunities that might still exist in the scope of CT education. Their literature review has identified 58 papers referring to CT definitions, course design, teaching methodology or assessing CT. Sondakh (2017) published a paper that provided the insight of CT in higher education, particularly focusing on assessing CT skills. She identified 22 relevant CT skills within 16 papers. It should be noted that the papers were selected with the intent to include the widest possible educational spectrum. Based on the detailed review of the four aforementioned literature reviews, our initial corpus of 135 relevant papers was ready for screening.

3.3. Inclusion criteria and paper screening

The inclusion criteria had to support the main purpose of this study that is to identify basic, domain independent dimensions of CT. After screening summary, introduction and discussion sections of each paper from the corpus of the initial 135 papers, the paper was included for CT concept analysis only if it was a CT definition proposal, CT taxonomy proposal or CT assessment. Moreover, the papers that applied the existing conceptual taxonomy to a specific domain or the papers that proposed a new conceptual taxonomy based on existing CT definition or existing CT taxonomy, were also included. In the second phase, CT concept analysis was performed to build a final paper corpus. Only papers that advocated conceptual dimensions were selected into final corpus. During the process, we also identified papers that built a conceptual taxonomy first that was later implemented within programming domain, where conceptual dimensions were translated to specific programming constructs. They were also included in the final corpus. A large number of papers were building on CT taxonomies specific to programming domain, but were not included in the final corpus. Among which, Brennan and Resnick (2012) was the most referenced paper. At the end of this process twenty-eight papers have been included in the final corpus.

4. Results

This section presents the results of the analysis of 28 papers that were included in the final corpus.

RQ1: What are the relevant theoretical, domain independent dimensions of CT?

The aim was to discover the widest possible range of general, domain independent CT concepts. Thirty-six distinct CT concepts were identified, with frequencies ranging from 23 to 1. CT concepts with the highest frequencies are presented in Table 1. Relevant CT dimensions are presented in CT Concept column, number of occurrences for each dimension is presented in Frequency column. The Reference column presents the list of papers where the particular dimension has been identified. It should be noted that different reference format is used to list the papers in the Reference column for greater legibility. The most common concepts are underlined: abstraction and algorithm.

As presented in Table 1, the abstraction concept and the algorithm concept have 23 occurrences within the literature. However, as can be observed from the reference column in Table 1, they were not identified within the same papers. The frequencies of other identified concepts are much lower than abstraction and algorithm.

CT Concept	Frequency	Reference
<u>Abstraction</u>	23	[3] [7] [9] [10] [15] [18] [22] [23] [27] [28] [33] [34] [38] [63] [64] [66] [70] [73] [74] [82] [83] [84] [90]
<u>Algorithm</u>	23	[3] [7] [9] [10] [13] [15] [18] [22] [23] [27] [28] [34] [38] [45] [49] [63] [64] [66] [70] [73] [74] [82] [90]
Decomposition	14	[3] [7] [18] [22] [23] [28] [38] [64] [66] [70] [73] [74] [82] [90]
Data representation	11	[7] [10] [13] [15] [27] [28] [66] [69] [73] [84] [90]
Modeling	9	[13] [15] [27] [28] [33] [34] [51] [69] [84]
Evaluation	8	[3] [13] [18] [27] [64] [69] [74] [82]
Generalization	8	[3] [18] [27] [28] [34] [64] [74] [90]

Table 1 – Identified CT concepts

For a better visualization of the frequency of CT concepts, Figure 1 shows a word cloud of all 36 distinct CT concepts.



Figure 1 – Word cloud of identified CT concepts

As can be observed from Figure 1, the algorithm concept and the abstraction concept are of the same, maximum size, while other identified concepts are smaller, according to their frequency of occurrence. Due to space limit, concepts with only one occurrence, such as Systematic analysis, Cooperation, Concretization, Collaboration, Information processing, Playfulness, Literacy, Computational Vocabulary, Linguistics, Scalability, Planning and Efficiency have reduced visibility.

RQ2: What is the proportion of intendent usage of relevant CT dimensions in the literature?

Table 2 presents the frequency of intendent usage of relevant CT dimensions identified in the literature. The most common identified purpose was CT definition, followed by CT assessment and CT taxonomy.

Intendent usage	Frequency	Reference
Definition	13	[7] [9] [10] [15] [18] [28] [33] [34] [49] [51] [70] [74] [82]
Assessment	10	[3] [13] [22] [27] [45] [63] [66] [69] [83] [90]
Taxonomy	5	[23] [38] [64] [73] [84]

Table 2 – Intendent usage of CT dimensions

5. Relevant computational thinking concepts

In this section we present the characteristics of relevant computational thinking concepts that were identified within our study. The concepts and their frequencies were presented in Table 1. Frequencies and paper references of intendent usage are presented at the end of description of each CT concept. However, this is by no means an exhaustive literature review of particular concepts, but rather an overview of their characteristics, scope and usage.

5.1. Abstraction

Abstraction can generally be characterized as the conceptual process of eliminating specificity by ignoring certain features. From a human ability perspective, abstraction is comprised of two complementary concepts: removing details to build simplification and deriving generalizations to illuminate essentials. Abstraction has been the focus of many studies within various disciplines, such as philosophy, psychology, cognitive science, mathematics and computer science. Whistler (2016) cites prominent philosophers who were debating how a reason evolves as an abstraction of thought. Abstract thinking was defined as the ability "to realistically imagine a problem and a solution" by developmental psychologist Piaget (1950). Piaget argued that children develop the ability to think abstractly, systematically and hypothetically as part of their last stage of cognitive development. Kramer (2007) asserted that abstraction is "fundamental to mathematics and engineering in general, playing a critical part in the production of models for analysis and in the production of sound engineering solutions" (Kramer, 2007, p.40). Frorer et al. (1997) addressed the complexity and various faces of abstraction in mathematics. Moreover, diSessa (2018) interprets mathematics as a "queen of abstract sciences". diSessa clarifies distinction between the mathematical (inferential) abstraction, abstraction in physic (empirical abstraction) and abstraction in computer science (practical abstraction). diSessa advocates that the skill of "peeling away" irrelevant particulars is not needed within the mathematical abstraction. This claim is further supported by Colburn and Shute (2007) who advocate that abstraction in "computer science is to be sharply distinguished with abstraction within mathematics". Abstraction is closely related to the modeling concept and the concept of generalization. The model is abstraction of a real or a conceptual complex system. Empirical sciences are normally concerned with two kinds of models: concrete models in the form of experimental apparatus and abstract model in the form of mathematics. The model is designed to expose significant features and characteristics of the system that is intended for the study, for the prediction or for the modification. Thus, a particular model represents only some of the many aspects of the complex system that could potentially be modeled. Namely, the aspects and the complexity of the particular model depend on the relevant level of abstraction that was employed by the model developer. Prusinkiewicz (1998) discusses the relevancy of abstraction levels in natural sciences. Moreover, Nielsen (2018) addresses the right degree of abstraction in the context of scientific knowledge and scientific abstraction. Nielsen argues that abstraction is equally as important as the scientific method, although it is relatively neglected in the science itself and in contemporary analytical philosophy of science. Computer science relates to various entities characterized as abstract or various activities characterized as abstraction, such as abstract data types, data abstraction, procedural and control abstraction. Lowe and Brophy (2017) claim that abstraction in computer science has taken on two different usages: the first use of abstraction relates to inheritance and uses the fewer details as an extension point for future functionality, while the other common use of abstraction is to hide away the details which are not required to understand a greater concept. The latter one is often conceptualized as encapsulation. This is further supported by Colburn and Shute

(2007) who advocate that information hiding is the primary objective of abstraction in computer science practices, such as programming languages, operating systems, network architecture, and design patterns. Within software engineering, abstraction involves the extraction of properties of an object according to some focus: only those properties are selected which are relevant with respect to the focus (Czarnecki, 1988). Michaelson (2017) presents several examples of programming practices concerning abstraction, such as the procedural abstraction, the functional abstraction and abstractions within arrays and recursion. Moreover, Michaelson links these programming practices with CT concepts. According to Wing (2011), abstraction process is "the most important and high-level thought process in computational thinking". Additionally, abstraction "is used to let one object stand for many. It is used to capture essential properties common to a set of objects while hiding irrelevant distinctions among them" (Wing, 2011). In the context of CT, layers of abstraction allows a problem solver to focus on one layer at the time, to define each layer and to illuminate relationships between layers (Wing, 2008). Furthermore, Csizmadia et al. (2015) define abstraction in the context of CT as the process of making an artefact more understandable through reducing the unnecessary detail. The core practice in CT abstraction is in choosing the right detail to hide that the problem becomes easier, without losing anything that is important. A key part is in choosing a good representation of the system. Different representations make different things easy to do (Csizmadia et al., 2015). Selby and Woollard (2014) cite various authors that observe abstraction in the context of developing the CT definition. Based on the literature review, they advocate that abstraction is the core concept that should be included in CT. Results of our analysis show 11 occurrences in CT definition category ([7] [9] [10] [15] [18] [28] [33] [34] [70] [74] [82]), 5 occurrences in CT taxonomy category ([23] [38] [64] [73] [84]) and 7 occurrences in CT assessment category ([9] [22] [27] [63] [66] [83] [90]).

5.2. Algorithms

We define algorithms as procedural building blocks of a computer programming, of a human thought and of a general problem solving. The purpose of this generalized definition is to manifest the multi-dimensionality of an algorithm concept. Within programming, algorithms are predefined procedures that are encoded with programming artefacts to incorporate sequencing, branching and looping. In the theoretical computer science, an algorithm is defined as a procedure that satisfies the criteria of finiteness, input, output, effectiveness, and definiteness (Knuth, 1997). In this context, algorithms are effectuated with formally defined (programming) languages, where statements are rigorously unambiguous. The aim is that such an algorithm becomes a machine executable artefact, characterized by Denning (2017) as "not any sequence of steps, but a series of steps that control some abstract machine or computational model without requiring human judgment" (Denning, 2017, p.33). Finally, Aho (2011) provides a definition of an algorithm in the context of Turing computability theory, stating that "The Turing machine provides a precise definition for the term algorithm: an algorithm for a function f is just a Turing machine that computes f ." (Aho, 2011, p.5). Mathematical algorithm is a computational procedure, a set of steps that takes an input, perform a finite number of allowed operations and produces an output. Among all the definitions of algorithms within mathematics, we emphasize the definition introduced by Fetzer (2001) that cites Kleene (1967), who defines algorithms as "effective procedures, which are rules or methods whose application to a problem within an appropriate domain leads invariably to a definite solution within a finite number of steps". Generally, the algorithmic nature of mathematics can be observed through proving many theorems that provide a finite procedure to answer a question or to calculate something (Basu et al., 2006). Algorithms are of great interest to many philosophers that advocate the relation of algorithms to cognition (Hill, 2015). In the philosophy of mind, the mental algorithm is a fragment in the computational theory of mind that treats human mind as information processing system. Fetzer (2001) cites prominent cognitive scientists and philosophers while he observes the concept of mental algorithm within computational and non-computational conception. Moreover, computational theory of mind is the foundation for cognitive theory proposed by Marr (2010), where cognitive processes consist of three layers: computational (semantic) level, algorithmic (syntax) level and implementational (physical) level. The algorithm design skills and effective usage of algorithms are fundamental practices in algorithmic problem solving. In educational context, these practices are advocated as algorithmic thinking. Knuth (1985) observed the concept of algorithmic thinking in mathematics, while Fustschek (2006) associates algorithmic thinking as one of the most important competencies that can be acquired by

learning informatics. Fustschek further defines algorithmic thinking as a pool of abilities that are concerned with the ability to analyze the given problems, the ability to specify a problem precisely, the ability to construct the correct algorithm, the ability to think about all possible circumstances to a given problem and the ability to improve the efficiency of an algorithm. Moreover, Fustschek advocates a strong creative aspect of the algorithmic thinking associated with the construction of new algorithms that solve new problems. In the context of CT, algorithmic thinking is concerned with getting to a solution through a clear definition of steps. It is the ability to think in terms of sequences and rules as a way of problem solving or understanding situations (Csizmadia et al., 2015). CT is associated with algorithmic thinking, pattern recognition and generalization in solving problems when repeatable patterns can be observed. According to our analysis, algorithms have 11 occurrences in CT definition category ([7] [9] [10] [15] [18] [28] [34] [49] [63] [70] [74]), 4 occurrences in CT taxonomy category ([23] [38] [64] [73]) and 8 occurrences in CT assessment category ([3] [13] [22] [27] [45] [63] [66] [90]).

5.3. Decomposition

Decomposition deals with breaking down a problem into smaller, more manageable components where each component can be managed independently. Problem decomposition practice exists in general scientific domain. Decomposition was advocated by Polya (1957) within mathematics as a part of his problem solving techniques called heuristics. Examples are to be found in the physics, where objects are decomposed into entities of hierarchy, like molecules, atoms and subatomic particles. In discrete mathematics, modular decomposition is a technique in several domains of combinatorics, such as the graphs and hypergraphs (Habib and Paul, 2010). In computer science, distinct variants of decomposition can be observed. Parnas (1972) investigated hierarchical decomposition in the context of modularity in order to decompose complex information system into a number of smaller, manageable modules. Moreover, decomposition is the crucial part of structured, object-oriented and functional programming paradigms. In terms of CT, Wing (2008) linked abstraction layers with hierarchical decomposition. Barr and Stephenson (2011) argue that decomposition is about breaking problems down into smaller parts that may be more easily solved. Decomposition is a way of thinking about problems, algorithms, artefacts, processes and systems in terms of their parts. These parts can then be understood, solved, developed and evaluated separately. This makes complex problems easier to solve and large systems easier to design (Curzon et al., 2014). Results of our analysis show there are 6 occurrences in CT definition category ([7] [18] [28] [70] [74] [82]), 4 occurrences in CT taxonomy category ([23] [38] [64] [73]) and 4 occurrences in CT assessment category ([3] [22] [66] [90]).

5.4. Data management

Data collection, data analysis and data representation are important aspects of the scientific research. However, in the context of CT, data is supporting computational thinking process by providing important information in the means of finding a data source, analyzing the data or using the data structures to represent data, such as graphs, charts, words or images (Conery et al., 2011). In our analysis, data representation concept has 4 occurrences in CT definition category ([7] [10] [15] [28]), 2 occurrences in CT taxonomy category ([73] [84]) and 5 occurrences in CT assessment category ([13] [27] [66] [69] [90]).

5.5 Modeling

Modeling is a core practice in science and a central part of scientific literacy. Schwarz et al. (2009) define scientific model as a representation that abstracts and simplifies a system by focusing on key features to explain and predict scientific phenomena. Working with scientific models involves constructing and using models, as well as evaluating and revising them. The development of sufficiently useful models typically requires a series of iterative "modeling cycles" where results of constructions and explanations activities are tested and revised repeatedly (Lesh and Harel, 2003). Modeling is particularly important in mathematics and engineering. Mathematical models are distinct from other categories of models mainly because they focus on structural characteristics of systems they describe, while models in physics, biology, or arts do not build upon the structural characteristics (Lesh and Harel, 2003). Kramer (2007) argues that modeling is the most important engineering technique, as models assist engineers to understand and analyze large and complex systems. Within

computing, modeling is an important and common form of representing different areas and levels of informatics and software engineering. In the theoretical computer science Aho (2011) defines a model of computation as "mathematical abstraction of a computing system" (Aho, 2011, p.4). Furthermore, he emphasizes Turing machine as the most important model of sequential computation. However, modeling within computing might not be confused with computational modeling, which is the use of computers to simulate and study the behavior of complex systems using mathematics and computational science. Computational modeling can be applied to any scientific domain, particularly in empirical sciences, such as physics, neurobiology, psychology and cognitive science (Stacewicz and Włodarczyk, 2010). Sabitzer and Pasterk (2015) observed modeling in the context of education, where competencies needed for the modeling process like abstraction, reduction, simplification, classification and generalization should be part of general education covering children of all ages. STEM educational researchers at K-12 level (Weintrop et al., 2016) have proposed CT taxonomy for mathematics and science, where, among other CT concepts, modeling and simulation concept is defined as designing, constructing, using and assessing computational models. According to our analysis, modeling concept has 5 occurrences in CT definition category ([15] [28] [33] [34] [51]), 1 occurrence in CT taxonomy category ([84]) and 3 occurrences in CT assessment category ([13] [27] [69]).

5.6. Evaluation

Evaluation generally refers to the process of determining the worth, merit or value of the subject that is exposed to evaluation – evaluand. The evaluation process normally involves some identification of relevant standards of worth, merit or value. Moreover, the process is also concerned with some investigation of the performance of the evaluands on these standards (Shaw et al., 2013). Distinction should be concerned between the term "assessment" and the term "evaluation" within educational domain. Sultana (2016) defines evaluation as the process for determining the success level of an individual based on data, while assessment is defined as the procedures or techniques used to collect the aforementioned data. In the context of CT, Curzon et al. (2014) propose evaluation as the process of ensuring an algorithmic solution is a good one: that it fits for purpose. Various algorithmic properties need to be evaluated including correctness, speed, whether they are economic in the use of resources and whether they are easy for people to use and to promote. There is a specific and often extreme focus on attention to detail in computational thinking based evaluation (Curzon et al., 2014). In our analysis, evaluation concept has 3 occurrences in CT definition category ([18] [74] [82]), 1 occurrence in CT taxonomy category ([64]) and 4 occurrences in CT assessment category ([3] [13] [27] [69]).

5.7. Generalization

Generalization can be observed from different aspects. In psychology, generalization is described as "the occurrence of relevant behavior under different, non-training conditions" (Stokes and Baer, 1977, p.350). From educational point of view, generalization is one of the fundamental activities in teaching and learning of mathematics (Hashemi et al., 2013). In software engineering, generalization is a crucial practice in object oriented programming, where generalization is the act of capturing similarities between classes and defining the similarities in a new, generalized class, called superclass. The ability to generalize and to abstract is an important concept in problem solving. Kamarudin et al. (2016) argue that generalization and abstraction provide more freedom in the construction of an idea than using a specific problem solving approach. Both concepts enable designers to initiate early moves in generating various ideas for problem solving. Within CT, generalization is associated with identifying patterns, similarities and connections. It is a way to quickly solve new problems based on previous solutions to problems, and a way of building on prior experience (Csizmadia et al., 2015). Selby and Woppard (2014) identify generalization as one of the most frequently occurring terms in CT literature. Moreover, they argued that generalization is used sparingly in the CT literature, but the idea of recognizing and reusing common parts of a solution or process is appropriate for inclusion in a definition of computational thinking. According to our analysis, generalization concept has 4 occurrences in CT definition category ([18] [28] [34] [74]), 1 occurrence in CT taxonomy category ([64]) and 3 occurrences in CT assessment category ([3] [27] [90]).

6. Discussion

The objective of this paper was to identify a basic set of domain independent dimensions of CT that researchers would agree upon. To date there is no consensus regarding the definitive or necessary components that would form the basic foundation of agreed CT concepts. According to the results presented in Table 1, abstraction and algorithms appear to be the prime prospects to build on consensus.

Results of our analysis in Table 1 are aligned with findings from previous works. Selby and Woolard (2014) conducted a research to identify a definition and a description of the core elements of CT. Based on the analysis, a consensus was found in the literature regarding abstraction concept and decomposition concept. Although consensus was not proposed for algorithmic design, the concept was found to be well defined across multiple disciplines and was included within CT definition. Araujo et al. (2016) found that algorithm and abstraction are the most commonly assessed CT skills. Lowe and Brophy (2017) argue that algorithms may be the most agreed CT concept across literature. Barr et al. (2011) report on consensus that emerged regarding the essential elements of CT during the meeting of a diverse group of educators with interest in CT from higher education, K-12 and industry. More than 82% of 697 respondents agreed or strongly agreed upon the essential elements of CT, where abstraction and algorithmic thinking were recognized as the essential concepts. Nevertheless, two literature review papers that were selected as the primary source of our study reveal comparably: Kalelioglu (2016) advocates that abstraction and algorithmic thinking are mostly used words in CT scope. Sondakh (2017) reports that abstraction and algorithms are the most commonly assessed CT concepts in higher education.

Results from Table 2 reveal that the main research focus is on defining and assessing CT skills and practices. This is aligned with the findings from Roman-Gonzalez et al. (2016) who claim that there is no consensus on formal definition of CT nor does a consensus exist concerning how to effectively measure CT, although both aspects are in the main focus of recent research.

To line up with general, domain independent CT concepts, papers that were addressing domain specific CT skills or practices were excluded from analysis. This is particularly true for programming, while the majority of identified papers were using taxonomies where components were associated with programming specific concepts and artefacts. However, the results from our study can be applied to any domain, including programming.

To expand upon our study, teaching and assessing CT should include abstraction and algorithms. In this context, how should an instrument for assessing these practices look like? According to findings from our study, it depends on the domain that would be the subject of the research. If the aim would be to assess the algorithmic abilities of introductory computer science students or the algorithmic abilities of a software engineer, the instrument should be constructed as conceptual intervention within pseudocode or as Parson problem design, in both cases with an emphasis on algorithms. On the other hand, if the aim would be to assess the algorithmic abilities of a primary school aged child, the instrument should be constructed as CS Unplugged task focusing on sorting and searching.

7. Future work

Further research is needed towards the consensus from necessary to definitive set of domain independent components that define CT. A Delphi study appears to be a proper starting point for this task. The purpose of the Delphi study is to "obtain the most reliable consensus of opinion of a group of experts" using surveys and questionnaires (Linstone and Turoff, 2002, p.10). To our best knowledge, there were two Delphi studies conducted in the context of CT: Kolodziej (2017) conducted the Delphi study in order to formally define higher education curriculum, while Rich and Langton (2016) used Delphi process to formalize CT definition. Rich and Logan included only programming specific CT practices, therefore their result is not relevant in domain independent context.

The results of our study support previous findings concerning the complex, multidimensional nature of computational thinking. The proficiency of a child, a student or a professional within CT shows signs of being more complex than a simple aggregation of his or her proficiency within particular CT

concept, such as abstraction or algorithms. In this context, further research is required to identify and evaluate significance of particular CT dimension within general problem solving proficiency of an individual. Several evaluation rubrics already exist in domain specific CT assessments, such as Dr. Scratch (Moreno-Leon et al., 2015), which is aimed to evaluate programming proficiency within the Scratch projects. Although the metric used by Dr. Scratch has been a subject of several evaluations Moreno-Leon et al. (2017) and Browning (2017), the utilized metrics treats each CT practice with equal significance. Thus, abstraction practice is equally important as the data representation practice. Moreover, the competencies level for "abstraction and decomposition" within Dr. Scratch should be evaluated with respect to theoretical computer science (Turner, 2018, p.149). From this perspective, competence level "definition of blocks" (Moreno-Leon et al., 2015, p.6) shows signs of being pure problem decomposition, or at least modularization and problem decomposition rather than abstraction and problem decomposition.

The role of problem solving competency requires further investigation in context of CT. On the one hand, results of our study identified problem solving as a domain independent CT component with 7 occurrences in the literature. On the other hand, Barr and Stephenson (2011) characterize CT as a distinct problem solving process that incorporates several CT components as well as several non-cognitive competencies, such as confidence and persistence.

Last but not least, the results of our study also addressed challenges regarding teaching and assessing aspects of CT. There appears to be a consensus among the computer science researchers and educators concerning the multidimensional structure of CT. As a consequence, learning abstraction, algorithms or decomposition stimulates proficiency level of CT. Moreover, as advocated by Roman-Gonzalez et al. (2017, p.154) "CT assessment is an extremely relevant and urgent topic to address", researchers should focus on assessing particular dimensions of CT. Finally, our investigation on relevant CT concepts revealed the "multi face" structure of particular CT concepts: the treatment of abstraction and algorithms is not exactly the same within mathematics, computer science or within cognitive sciences. As CT is positioned as the umbrella of these "multi faced" concepts, further research is needed on possible "multi face" structure of computational thinking itself.

8. References

- [1] Aho A.M. (2011) What is Computation? Computation and Computational Thinking. *Ubiquity*, an ACM publication. January, 2011, ACM New York, NY, USA, 1-8.
- [2] Araujo A.L.S.O., Andrade W.L. and Guerrero D.D.S. A systematic mapping study on assessing computational thinking abilities. 2016 IEEE Frontiers in Education Conference (FIE), 1-9.
- [3] Araujo A.L.S.O., Santos J.S., Andrade W.L., Guerrero D.D.S and Dagiene V. (2017) Exploring Computational Thinking Assessment in Introductory Programming Courses. 2017 IEEE Frontiers in Education Conference (FIE), 1-9.
- [4] Barate A., Ludovico L.A. and Malchiodi D. (2017) Fostering Computational Thinking in Primary School through a LEGO-based Music Notation. International Conference on Knowledge Based and Intelligent Information and Engineering Systems, KES2017, 6-8 September 2017, Marseille, France, 1334-1344.
- [5] Barbara Sabitzer B., and Pasterk S. (2015) Modeling: A computer science concept for general education. 2015 IEEE Frontiers in Education Conference (FIE), 1-5.
- [6] Barr D., Harrison J. and Conery L. (2011) Computational Thinking: A Digital Age. *Learning & Leading with Technology*, March/April 2011, 20-23.
- [7] Barr V. & Stephenson C. (2011) Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community? Magazine ACM Inroads archive Volume 2 Issue 1, March 2011. ACM New York, NY, USA, 48-54.
- [8] Basu S., Pollack R. and Roy M-R. (2006) Algorithms in Real Algebraic Geometry, Second edition. Springer-Verlag Berlin Heidelberg 2003, 2006, 281-291.

- [9] Billionniere E. V. (2011) Assessing Cognitive Learning of Analytical Problem Solving. Doctoral dissertation, Arizona State University, December 2011.
- [10] Bort H. and Brylow D. (2013) CS4Impact: Measuring Computational Thinking Concepts Present in CS4HS Participant Lesson Plans. Proceeding of the 44th ACM technical symposium on Computer science education, Denver, Colorado, USA — March 06 - 09, 2013, ACM New York, NY, USA, 427-432.
- [11] Brennan K. and Resnick M. (2012) New frameworks for studying and assessing the development of computational thinking. Annual American Educational Research Association meeting 2012, Vancouver, BC, Canada, 1-25.
- [12] Browning S.F. (2017) Using Dr. Scratch as a Formative Feedback Tool to Assess Computational Thinking. Master thesis, Brigham Young University 2017.
- [13] Chen G., Shen J., Barth-Cohen L., Jiang S., Huang X. and Eltoukhy M. (2017) Assessing elementary students computational thinking in everyday reasoning and robotics programming. Computers and Education 109, 162-175.
- [14] Colburn T. and Shute G. (2007) Abstraction in Computer Science. Minds & Machines (2007) 17, 169-184.
- [15] Committee for the Workshops on Computational Thinking, Computer Science and Telecommunications Board, Division on Engineering and Physical Sciences (2010) Report of a Workshop on the Scope and Nature of Computational Thinking (2010). The National Academies Press, Washington, D.C.
- [16] Conery L.S., Stephenson C., Barr D., Barr V., Harrison J., James J. and Sykora C. (2011) Computational Thinking in K-12 Education teacher resources, Second edition.
- [17] Csizmadia, A., Curzon, P., Dorling, M., Humphreys, S., Ng, T., Selby, C., and Woppard, J. (2015) Computational thinking: A guide for teachers. Computing at school.
- [18] Curzon P., Dorling M., Ng T., Selby C. and Woppard J. (2014) Developing computational thinking in the classroom: a framework. Computing At School, 2014.
- [19] Czarnecki K. (1998) Generative Programming - Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Doctoral dissertation, Technical University of Ilmenau, Germany, October 1998.
- [20] Denning P.J. (2017) Remaining Trouble Spots with Computational Thinking. Magazine Communications of the ACM, Volume 60 Issue 6, June 2017. ACM New York, NY, USA, 33-39.
- [21] diSessa A. (2018) Computational Literacy and "The Big Picture" Concerning Computers in Mathematics Education, Mathematical Thinking and Learning, 20:1, 3-31.
- [22] Djambong T., Freiman V., Gauvin S., Paquet M. and Chiasson M. (2018) Measurement of Computational Thinking in K-12 Education: The Need for Innovative Practices. Digital Technologies: Sustainable Innovations for Improving Teaching and Learning. Springer International Publishing AG 2018, 193-222.
- [23] Dong Y., Catete V., Jocius R., Lytle N., Barnes T., Albert J., Joshi D., Robinson R. and Andrews A. (2019) PRADA: A Practical Model for Integrating Computational Thinking in K-12 Education. Proceedings of the 50th ACM Technical Symposium on Computer Science Education, February 27–March 2, 2019, Minneapolis, MN, USA. 2019 Association for Computing Machinery, 906-912.
- [24] Fetzer, J.H. (2001) Computers and Cognition: Why Minds Are Not Machines. Springer Science+Business Media Dordrecht, 101-130.
- [25] Frorer P., Hazzan O. and Manes M. (1997) Revealing the faces of abstraction. International Journal of Computers for Mathematical Learning 2, 1997, 217-228.

- [26] Futschek, G. (2006) Algorithmic Thinking: The Key for Understanding Computer Science. In Lecture Notes in Computer Science 4226, Springer, 159-168.
- [27] Gouws L., Bradshaw K. and Wentworth P. (2013) First Year Student Performance in a Test for Computational Thinking. Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference, ACM New York, NY, USA, 271-277.
- [28] Grover S. and Pea R. (2013) Computational Thinking in K-12: A Review of the State of the Field. *Educational Researcher*, Vol. 42 No. 1, 38-43.
- [29] Habib M. and Paul C. (2010) A survey of the algorithmic aspects of modular decomposition. *Computer Science Review*, Volume 4, Issue 1, February 2010, 41-59.
- [30] Hashemia N., Abua M.S., Kashefia H. and Rahimib K. (2013) Generalization in the Learning of Mathematics. 2nd International Seminar on Quality and Affordable Education (ISQAE 2013), 208-215.
- [31] Hill R.K. (2015) What an Algorithm Is. *Philos. Technol.* (2016) 29. Springer Science+Business Media Dordrecht 2015, 35-59.
- [32] Hsu T-C., Chang S-C. and Hung Y-T (2018) How to learn and how to teach computational thinking: Suggestions based on a review of the literature. *Computers & Education* 126 (2018), 296-310.
- [33] Ingram-Goble A. (2013) Playable stories: making programming and 3d role-playing game design personally and socially relevant. Doctoral dissertation, Indiana University, October 2013.
- [34] ISTE & CSTA. (2011) Operational Definition of Computational Thinking for K-12 Education.
- [35] James Lockwood J. and Mooney A. (2018) Computational Thinking in Secondary Education: Where does it fit? A systematic literary review. *International Journal of Computer Science Education in Schools*, Jan 2018, Vol. 2, No. 1, 1-20.
- [36] Kalelioglu F., Gulbahar Y. and Kukul V. (2016) A Framework for Computational Thinking Based on a Systematic Research Review. *Baltic Journal of Modern Computing*, Vol. 4 (2016), No. 3, 583-596.
- [37] Kamarudin K.M., Ridgwaya K. and Ismail N. (2016) Abstraction and Generalization in Conceptual Design Process: Involving Safety Principles in TRIZ-SDA Environment. *Procedia CIRP* 39 (2016), 16-21.
- [38] Kao E. (2011) Exploring Computational Thinking at Google. *The Voice of K-12 Computer Science Education and its Educators*, Volume 7, Issue 2, May 2011, 6-7.
- [39] Khine M. (2018) Computational Thinking in the STEM Disciplines, Foundations and Research Highlights. Springer International Publishing AG, part of Springer Nature 2018.
- [40] Kitchenham B. and Charters S. (2007) Guidelines for performing Systematic Literature Reviews in Software Engineering Version 2.3. *Engineering*, Volume 45, Issue 4.
- [41] Kleene, S.C. (1967). Mathematical Logic. New York: John Wiley and Sons.
- [42] Knuth D.E. (1985) Algorithmic Thinking and Mathematical Thinking, *The American Mathematical Monthly*, 92:3, 170-181.
- [43] Knuth D.E. (1997) The Art of Computer Programming: Volume 1: Fundamental Algorithms, Third Edition, Addison-Wesley, 1997.
- [44] Kolodziej M. (2017) Computational Thinking In Curriculum For Higher Education. Doctoral dissertation, Pepperdine University, May 2017.
- [45] Korkmaz O., Cakir R. and Ozden M. Y. (2017) A validity and reliability study of the computational thinking scales (CTS). *Computers in Human Behavior* 72 (2017), 558-569.

- [46] Kramer J. (2007) Is abstraction the key to computing? Communications of the ACM April 2007/Vol. 50, No. 4, 37-42.
- [47] Lesh R. and Harel G. (2003) Problem Solving, Modeling, and Local Conceptual Development, Mathematical Thinking and Learning, 5:2-3, 157-189.
- [48] Linstone H.A. and Turoff M. (2002) The Delphi Method Techniques and Applications. <https://web.njit.edu/~turoff/pubs/delphibook/delphibook.pdf>, accessed 10.6.2019.
- [49] Lishinski A., Yadav A., Enbody R. and Good J. (2016) The Influence of Problem Solving Abilities on Students' Performance on Different Assessment Tasks in CS1. Proceedings of the 47th ACM Technical Symposium on Computing Science Education, Memphis, Tennessee, USA — March 02 - 05, 2016, ACM New York, NY, USA, 329-334.
- [50] Lowe T. and Brophy S. (2017) An Operationalized Model for Defining Computational Thinking. 2017 IEEE Frontiers in Education Conference (FIE), 1-8.
- [51] Marcelino M. J., Pessoa T., Vieira C., Salvador T. and Mendes A. J. (2018) Learning Computational Thinking and scratch at distance. Computers in Human Behavior 80 (2018), 470-477.
- [52] Marr D.C. (2010) Vision: A Computational Investigation into the Human Representation and Processing of Visual Information. Mit Press 2010.
- [53] Michaelson G. (2017) From Problems to Programs with Computational Thinking. <http://www.macs.hw.ac.uk/~greg/Teaching%20Programming/From%20problems%20to%20programs%20with%20CT.pdf>, accessed 10.6.2019.
- [54] Moreno-Leon J., Carlos R.J., Harteved C., Roman-Gonzalez M. and Robles G. (2017) On the Automatic Assessment of Computational Thinking Skills: A Comparison with Human Experts. Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems, 2788-2795.
- [55] Moreno-Leon J., Robles G. and Roman-Gonzalez M. (2015) Dr. Scratch: Automatic Analysis of Scratch Projects to Assess and Foster Computational Thinking. RED-Revista de Educacion a Distancia. Numero 46 15-Sep-2015, 1-23.
- [56] National Research Council (2011) Report of a workshop of pedagogical aspects of computational thinking. Washington, D.C. National Academies Press.
- [57] Nielsen N. (2018) Scientific Knowledge and Scientific Abstraction. <https://medium.com/@jnnielsen/scientific-knowledge-and-scientific-abstraction-4adeb6589706>, accessed 10.6.2019.
- [58] Papert S. (1980) Mindstorms: children, computers, and powerful ideas. Basic Books, Inc USA.
- [59] Parnas, D.L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM. 15(12), 1972, 1053-1058.
- [60] Piaget J. (1950) The psychology of intelligence. London, UK: Routledge.
- [61] Polya, G. (1957). How to Solve It: A New Aspect of Mathematical Method 2nd edn (Garden City, NJ, Doubleday).
- [62] Prusinkiewicz P. (1998) In search of the right abstraction: The synergy between art, science, and information technology in the modeling of natural phenomena. C. Sommerer and L. Mignonneau (Eds.): Art @ Science. Springer, Wien, 1998, 60-68.
- [63] Qualls J.A., Grant M.M. and Sherrell L.B. (2011) CS1 students' understanding of computational thinking concepts. Journal of Computing Sciences in Colleges, Volume 26 Issue 5, May 2011, Consortium for Computing Sciences in Colleges, USA, 62-71.

- [64] Rad P., Roopae M., Beebe N., Shadaram M., Au Y. A. (2018) AI Thinking for Cloud Education Platform with Personalized Learning. Proceedings of the 51st Hawaii International Conference on System Sciences, 2018, 3-12.
- [65] Rich P.J. and Langton M.B. (2016) Computational Thinking: Toward a Unifying Definition. J.M. Spector et al. (eds.), Competencies in Teaching, Learning and Educational Leadership in the Digital Age. Springer International Publishing Switzerland 2016, 229-242.
- [66] Rodriguez B., Kennicutt S., Rader C. and Camp T. (2017) Assessing Computational Thinking in CS Unplugged Activities. Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, Washington, USA — March 08 - 11, 2017, ACM New York, NY, USA, 501-506.
- [67] Roman-Gonzalez M., Moreno-Leon J. and Robles G. (2017) Complementary Tools for Computational Thinking Assessment. Conference Proceedings of International Conference on Computational Thinking Education 2017. Hong Kong: The Education University of Hong Kong, 154-159.
- [68] Roman-Gonzalez M., Perez-Gonzalez JC. and Jimenez-Fernandez C. (2016) Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test. Computers in Human Behavior 72 (2017), 678-691.
- [69] Romero M., Lepage A. and Lille B. (2017) Computational thinking development through creative programming in higher education. International Journal of Educational Technology in Higher Education (2017), 1-15.
- [70] Rowe E., Cunningham K., Gasca S. (2017) Assessing Implicit Computational Thinking in Zoombinis Gameplay: Pizza Pass, Fleens & Bubblewonder Abyss. Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play, October 15–18, 2017, Amsterdam, Netherlands, 195-200.
- [71] Sabitzer B., Demarle-Meusel H. and Jarnig M. (2018) Computational Thinking Through Modeling In Language Lessons. 2018 IEEE Global Engineering Education Conference (EDUCON), 1913-1918.
- [72] Schwarz, C. V., Reiser, B. J., Davis, E. A., Kenyon, L., Acher, A., Fortus, D., Schwartz Y., Hug B. and Krajcik J. (2009). Developing a learning progression for scientific modeling: Making scientific 147 modeling accessible and meaningful for learners. Journal of Research in Science Teaching, 46(6), 632-654.
- [73] Seiter L. and Foreman B. (2013) Modeling the Learning Progressions of Computational Thinking of Primary Grade Students. Proceedings of the ninth annual international ACM conference on International computing education research, ACM New York, NY, USA, 59-66.
- [74] Selby C. and Woppard J. (2014) Refining an Understanding of Computational Thinking. University of Southampton Institutional Repository.
- [75] Shaw I., Greene J.C., Mark M.M. (2013) The SAGE Handbook of Evaluation. SAGE Publications Ltd.
- [76] Shute V.J., Sun C. and Asbell-Clarke J. (2017) Demystifying computational thinking. Educational Research Review 22 (2017), 142-158.
- [77] Simon Sultana, S. (2016) Defining The Competencies, Programming Languages, And Assessments For An Introductory Computer Science Course. Doctoral dissertation, Old Dominion University, August, 2016.
- [78] Sondakh D.E. (2017) Review of Computational Thinking Assessment in Higher Education. ResearchGate,
https://www.researchgate.net/profile/Debby_Sondakh/publication/324984840_Review_of_ComputationalThinkingAssessment_in_Higher_Education/links/5af0378aa6fdcc85

- [79] Stacewicz P. and Włodarczyk A. (2010) Modeling in the Context of Computer Science - A Methodological Approach. *Studies in Logic, Grammar And Rhetoric* 20 (33) 2010.
- [80] Stokes T. and Baer D. (1977) An implicit technology of generalization. *Journal of Applied Behavior Analysis*, 10, 349-367.
- [81] Turner R. (2018) Computational Artifacts Towards a Philosophy of Computer Science. Springer-Verlag GmbH Germany, part of Springer Nature 2018.
- [82] Walliman G. (2015) Genost: A System for Introductory Computer Science Education with a Focus on Computational Thinking. Doctoral dissertation, Arizona State University, May 2015.
- [83] Webb H. C. and Rosson M. B. (2013) Using Scaffolded Examples to Teach Computational Thinking Concepts. Proceeding of the 44th ACM technical symposium on Computer science education, Denver, Colorado, USA - March 06 - 09, 2013. ACM New York, NY, USA, 95-100.
- [84] Weintrop D., Beheshti E., Horn M., Orton K., Jona K., Trouille L. and Wilensky U. (2016) Defining Computational Thinking for Mathematics and Science Classrooms, *Journal of Science Education and Technology*, February 2016, Volume 25, Issue 1, 127-147.
- [85] Werner L., Denner J. and Campe S. (2012) The fairy performance assessment: Measuring computational thinking in middle school. Proceedings of the 43rd ACM technical symposium on Computer Science Education, Raleigh, North Carolina, USA — February 29 - March 03, 2012, ACM New York, NY, USA, 215-220.
- [86] Whistler D. (2017) Abstraction and Utopia in Early German Idealism. *Russian Journal of Philosophy & Humanities* Volume 1 #2 2017, 3-22.
- [87] Wing J.M. (2006) Computational thinking. *Communications of the ACM - Self managed systems CACM*, Volume 49 Issue 3, March 2006. ACM New York, NY, USA, 33-35.
- [88] Wing J.M. (2008) Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society* 2008, 3717-3725.
- [89] Wing J.M. (2011) Research Notebook: Computational Thinking--What and Why? The Link. The magazine of the Carnegie Mellon University School of Computer Science. <https://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why>, accessed 10.6.2019.
- [90] Yagci M. (2018) A valid and reliable tool for examining computational thinking skills. *Education and Information Technologies*, Springer Science+Business Media, LLC, part of Springer Nature 2018, 929-951.

Toward meaningful algorithmic music-making for non-programmers

Matt Bellingham

School of Performing Arts
University of Wolverhampton
matt.bellingham@wlv.ac.uk

Simon Holland

Music Computing Lab
The Open University
s.holland@open.ac.uk

Paul Mulholland

Knowledge Media Institute
The Open University
p.mulholland@open.ac.uk

Abstract

Algorithmic composition typically involves manipulating structural elements such as indeterminism, parallelism, choice, multi-choice, recursion, weighting, sequencing, timing, and looping. There exist powerful tools for these purposes, however, many musicians who are not expert programmers find such tools inaccessible and difficult to understand and use. By analysing a representative selection of user interfaces for algorithmic composition, through the use of the Cognitive Dimensions of Notations (CDN) and other analytical tools, we identified candidate design principles, and applied these principles to create and implement a new visual formalism, programming abstraction and execution model. The resulting visual programming language, Choosers, is designed to allow ready visualisation and manipulation of structural elements of the kind involved in algorithmic music composition, while making minimal demand on programming ability. Programming walkthroughs with novice users were used iteratively to refine and validate diverse aspects of the design. Currently, workshops with musical experts and teachers are being conducted to explore the value of the language for varied pragmatic purposes by expressing, manipulating and reflecting on diverse musical examples.

Introduction

Algorithm composition can be defined as the ‘partial or total automation of music composition by formal, computational means’ (Fernández and Vico, 2013), and typically involves structural elements such as indeterminism, parallelism, choice, multi-choice, recursion, weighting, and looping (Baratè, 2008). At PPIG 2014 (Bellingham et al., 2014) we presented a review of existing tools, such as Max (Puckette, 1991) and SuperCollider (McCartney, 2002), for manipulating these and other elements of music. This review, using the Cognitive Dimensions of Notations framework (Green and Petre, 1996), resulted in the following findings. First, we found that most existing software requires the user to have a considerable understanding of programming constructs—represented either graphically (e.g Max, Pure Data) or textually (e.g. SuperCollider, ChucK, Csound): such constructs require a significant learning overhead. Second, in some software, users are required to have an understanding of musical notation and/or music production equipment such as mixing desks and patchbays. Third, several systems imposed working practices un conducive to compositional processes. Fourth, in some cases the user was unable to define, or subsequently change, the musical structure. Finally, complex visual design in graphical programming languages led to patches with multiple connections, making them difficult to read and navigate. These findings led to the development of a prototype visual programming language (Bellingham et al., 2017) designed to allow structural elements of the kind involved in algorithmic music composition to be readily visualised and manipulated, while making little or no demand on programming ability. This system, called Choosers, centres around a novel non-standard programming abstraction (the Chooser) which controls indeterminism, parallelism, choice, multi-choice, recursive data, weighting, and looping. We have performed two programming walkthroughs (Bellingham et al., 2018) to test the ability of self-taught music producers without programming skills to use Choosers to carry out a range of rudimentary algorithmic composition tasks; to identify usability and user experience problems in the current design; and to identify tensions and trade-offs in the interaction design of the system. These experiments were carried out using a Wizard of Oz interface supported by a fully implemented back-end written in SuperCollider. Through this empirical work we demonstrated the ability of non-programmers to work with this type of notation. We are now interested in understanding the range of musical material that can be expressed using Choosers.

This paper considers the ability of Choosers to represent musically meaningful pieces while promoting structural malleability at the surface. To enable understanding of the more musically meaningful examples that form the central part of the paper, we first present a brief overview of the principal elements of Choosers.

Choosers; annotated practical examples

Sound samples are shown in boxes, and can be auditioned by clicking on them. Samples can be assembled into sequences using arrows, shown in Figure 1. Samples in a sequence play in the order indicated by the direction of the arrows. Only a single arrow can enter or exit each element in a sequence. This deliberate limitation reflects the fact that parallelism and choice are dealt with elsewhere in the language. Boxes and sequences can be put inside other boxes, thereby packaging them into a single unit.

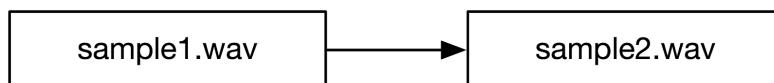


Figure 1 - Samples shown in boxes, and sequence shown using arrows.

Soundable Chooser

Boxes referring to samples or sequences can be snapped together vertically to create what are known as Choosers. Figure 2 shows a Chooser with two lanes, each containing a sample. The number in the nose cone indicates that, at run time, just one of the lanes will be selected at random (subject to the restrictions described below). On different runs, different choices may be made. By manipulating the number in the nose cone, any number of lanes from 0 to 2 can be chosen randomly to play simultaneously. A Chooser can have any number n of lanes. By manipulating the number in the nose cone, any number of lanes from 0 to n can be chosen randomly at run time and played simultaneously. Each lane has a weight associated with it; in Figure 2, changing the weight of one of the lanes to 2 would result in that lane being twice as likely to be chosen as the other. Any sample can be set to loop indefinitely when selected on a particular run, or to play just once by turning the lane's loop setting on or off. Alternatively, a finite number of loops can be specified by adding a number inside the loop icon. Indefinite looping of a single sample may not always be desired, and for this reason we now introduce Time Choosers.

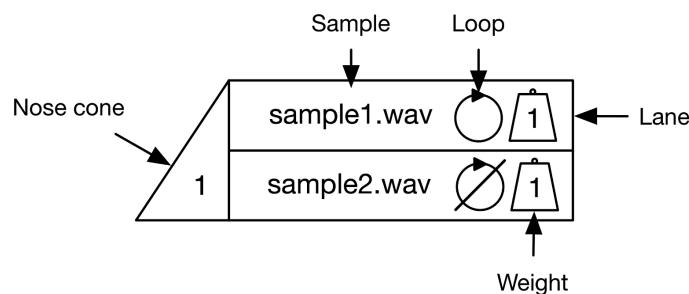


Figure 2 - A Soundable Chooser which allows control over indeterminism, parallelism, choice, multi-choice, weighting, and looping.

Time Choosers and Full Choosers

Figure 3 shows a Time Chooser, containing a duration (typically measured in bars or beats), attached to the bottom of a Soundable Chooser. This creates a Full Chooser, or just a Chooser for short. When the Full Chooser shown in Figure 3 is played, if the looping drums or bass samples are chosen on a given run, they will not play indefinitely but will be cut off after 4 bars by the Time Chooser. If the Time Chooser duration is cleanly divisible by the sample duration, every repetition of the sample will run to completion, but if the Time Chooser duration is not cleanly divisible by the sample duration

(for example, if the bass sample in Figure 3 had a duration of 3 bars), the Time Chooser would cut playback mid-sample. This is called a hard stop.

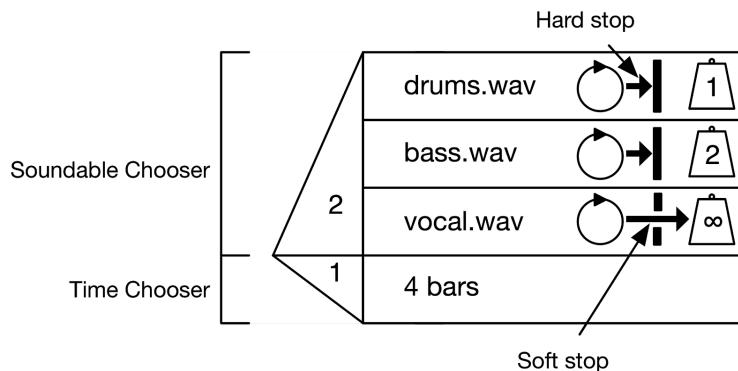


Figure 3 - A Full Chooser, consisting of a Soundable Chooser on the top and a Time Chooser on the bottom. The duration of the Soundable Chooser is moderated by the Time Chooser.

In Figure 3, the drums and bass samples are set to a hard stop, and the vocal sample is set to a soft stop. In contrast to a hard stop, when a Time Chooser's duration has elapsed, a chosen lane with a soft stop will continue to play until the end of its current loop. A Time Chooser can be used alone as part of a sequence—however, when used in this way it will simply result in a rest of the specified duration. A Time Chooser's nose cone can be set to either one or zero. If set to one, a single time lane will be chosen at run time. If it is set to zero, no time lanes will be selected and the Soundable Chooser will run as though there is no Time Chooser. This allows for quick low viscosity changes of arrangement, with the possibility of infinite playback if the Soundable Chooser lanes are set to loop. If the Soundable Chooser is not set to loop, the sample(s) will play and the Chooser will be released when they have finished playing, regardless of length.

Choosers uses ‘infinity’ as a maximal setting in three contexts; weight, time-lane duration, and Soundable Chooser nose cone. The effect of infinite weight, as seen in Figures 3 and 4, can be explained using a ‘priority boarding’ analogy. In Figure 3, the vocal sample has infinite weight and so must be the first selection, leaving one further selection to be made between the drums and bass samples. If the nose cone number is lower than the number of infinite-weighted lanes then the selection will only occur between the infinite-weighted lanes, and each lane has an equal weighting. If a lane has a weight of zero it has ‘no ticket’ and cannot be selected, regardless of the nose cone number. Infinite duration in a time lane is an alternative to running a Soundable Chooser with no corresponding Time Chooser. Finally, when used in a Soundable Chooser’s nose cone, infinity will select all available lanes.

We will now present examples of Choosers in practice to explore the range of musical structures that can be expressed and how musically meaningful variations can be made and auditioned in range of different genres. In an attempt to avoid dancing about architecture we offer audio from the examples in this paper¹.

Example 1

Figure 4 shows a simple example in the rock genre. The sequence shown in Figure 4 is populated by the named Choosers shown underneath them; note that Choosers may be named via a fin at the top of each Chooser, with these names used to refer to them elsewhere in the sequence. Thus, the sequence shown at the top of Figure 4 sounds the same as a sequence created by drawing a line directly between the two Choosers. Note that a named Chooser can be referenced from multiple locations in a visual program, meaning that changes to the named Chooser will be reflected across all referents.

¹ <https://figshare.com/s/d2edaf6486812ff1f596>

Multiplication notation (e.g. ‘ $\times 2$ ’, as seen in the sequence at the top of Figure 4) is very commonly used and understood by musicians (e.g. lead sheets, chord charts) and it allows for low-viscosity auditioning.

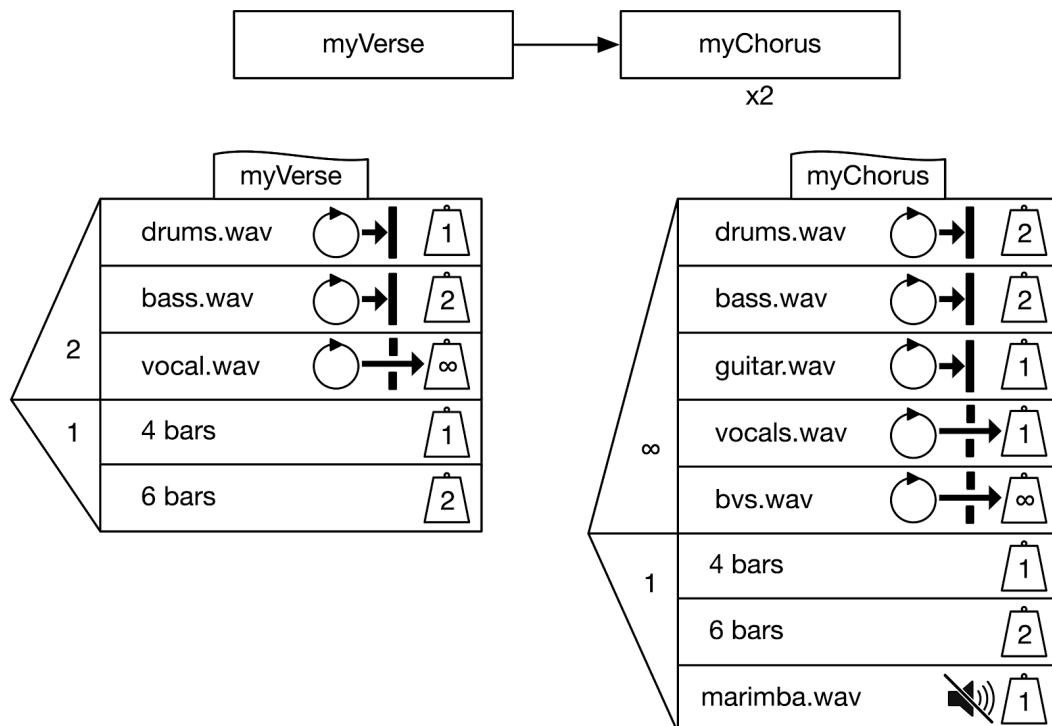


Figure 4 - An example showing sequence, nesting, naming via a fin at the top of the Choosers, and nondeterministic duration.

In a classic paper on the nature of musical creativity and algorithmic music composition, Bruce Jacob (1996) considers the ‘hard work’ of iteratively trying various options before making a final choice. A goal of our system is to support the user during this ‘hard work’ phase to make quick, impactful, and non-destructive changes, and to audition the results. By limiting options, surfacing key parameters, and maximising combinatorial usage, the possible interactions between the settings of Soundable and Time Choosers can make the results more varied than might be imagined. In precision-timed rock examples, the different effects of hard and soft stops provide a wide range of musical results. For example, consider the Chooser named ‘myVerse’ in Figure 4; the Soundable Chooser nose cone is currently set to 2, meaning that two of the three lanes will be selected when it is run. If the number were set to 0 then no lanes would be selected, but as the Time Chooser’s nose cone is set to 1 then a duration will be selected and the Chooser will run silently for that duration. If the Soundable Chooser nose cone is set to 2 and the Time Chooser nose cone is set to 0 then no duration will be selected, meaning that the Soundable Chooser will run without being stopped prematurely. If both nose cones are set to 0 then the Chooser will be skipped. Note that the Time Chooser nose cone can only be set to either 0 or 1; multiple durations cannot be selected. A soundable file can be used in a Time Chooser lane and, if selected, the sample’s length will become the duration used by the Time Chooser. An example of this can be seen on the lower right of Figure 4; a marimba sample is used in a lane in the Time Chooser. The sample can be used for duration only, or for both duration control and playback, by setting the sample to play or mute via the speaker icon.

Example 2

When there is one or more levels of nesting then considerably more complex structures and meaningful modifications become possible. The next example, shown in Figure 5, shows a piece of music in which the Chooser named ‘top’ refers to the two instruments (piano and violin) which are, in turn, populated by named Choosers. The piano is split into ‘lefthand’ and ‘righthand’ Choosers, which

each select one of two possible samples. Note that the ‘lefthand’ and ‘righthand’ child Choosers are set to not loop, while the ‘piano’ Chooser is set to loop. If we focus on ‘lefthand’, we can see that the ‘piano’ Chooser will trigger the choice of one of two samples for playback. Once complete, the ‘lefthand’ Chooser will be triggered again due to the loop setting of the ‘piano’ Chooser, with the potential for a different sample choice. An alternative would be for the ‘lefthand’ Chooser’s lanes to be set to loop; this would result in the choice of one sample which would then loop continuously.

The Choosers shown in Figure 5 can be manipulated to yield significant musical changes, such as:

- Changing the nose cone of ‘top’ or ‘piano’ to 1 will result in the selection of just one lane. Alternatively, setting the weight of a lane to 0 will remove it from the selection.
- The ‘...hand’ Choosers can be set to loop, and the corresponding lanes in the ‘piano’ Chooser set to not loop. This will result in the selection and looping of one sample, rather than continual reselection. Additionally, the nose cone values of the ‘...hand’ Choosers can be set to 0 (no selection) or 2 or infinity (playback of both samples simultaneously). The same techniques can be used with the ‘violin’ Chooser.

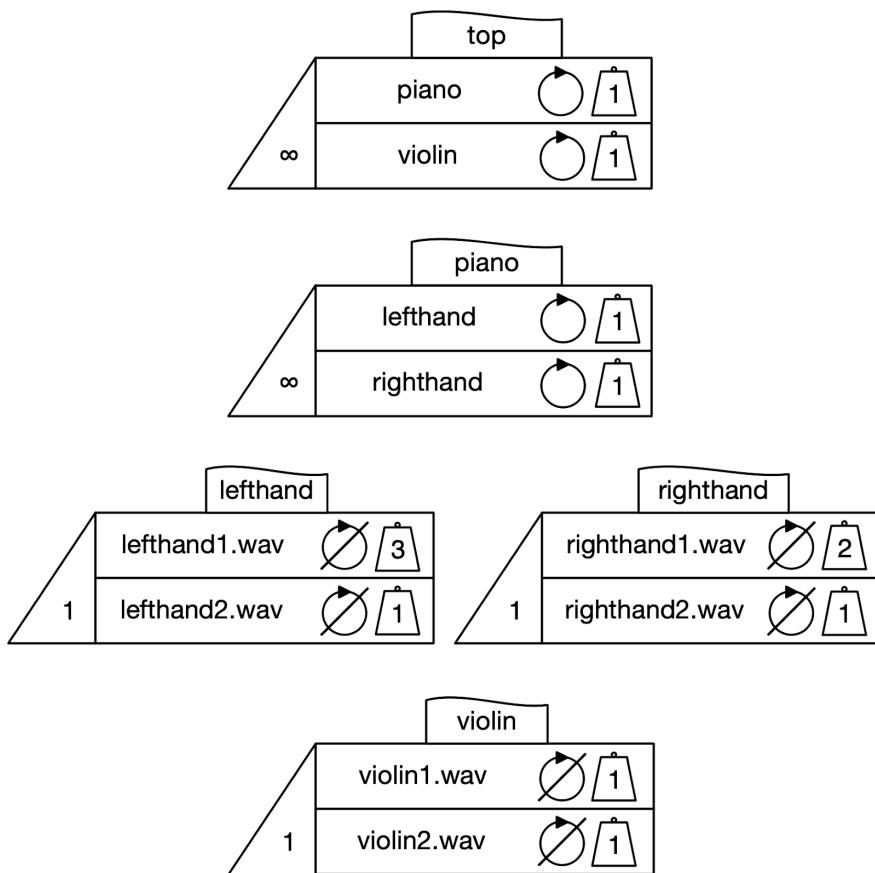


Figure 5 - Piano and violin example. The ‘lefthand’ pattern is three beats long and the ‘righthand’ pattern is four beats long, meaning that they will align every twelve beats. The ‘violin’ sample is eight beats long.

A central aspect of music is timing. By adding Time Choosers to the nested structure shown in Figure 5 we are able to manipulate time at different levels; an example is shown in Figure 6. Here we see that the ‘lefthand’, ‘righthand’, and ‘violin’ Choosers have been converted into Full Choosers by the addition of Time Choosers, with the durations mirroring the duration of the samples. Note that the ‘violin’ Chooser now has an added blank lane in the Soundable Chooser with a weight of 2; if selected a blank lane will ‘play’ silently for the duration set by the Time Chooser, which in this case will introduce a rest of 8 bars.

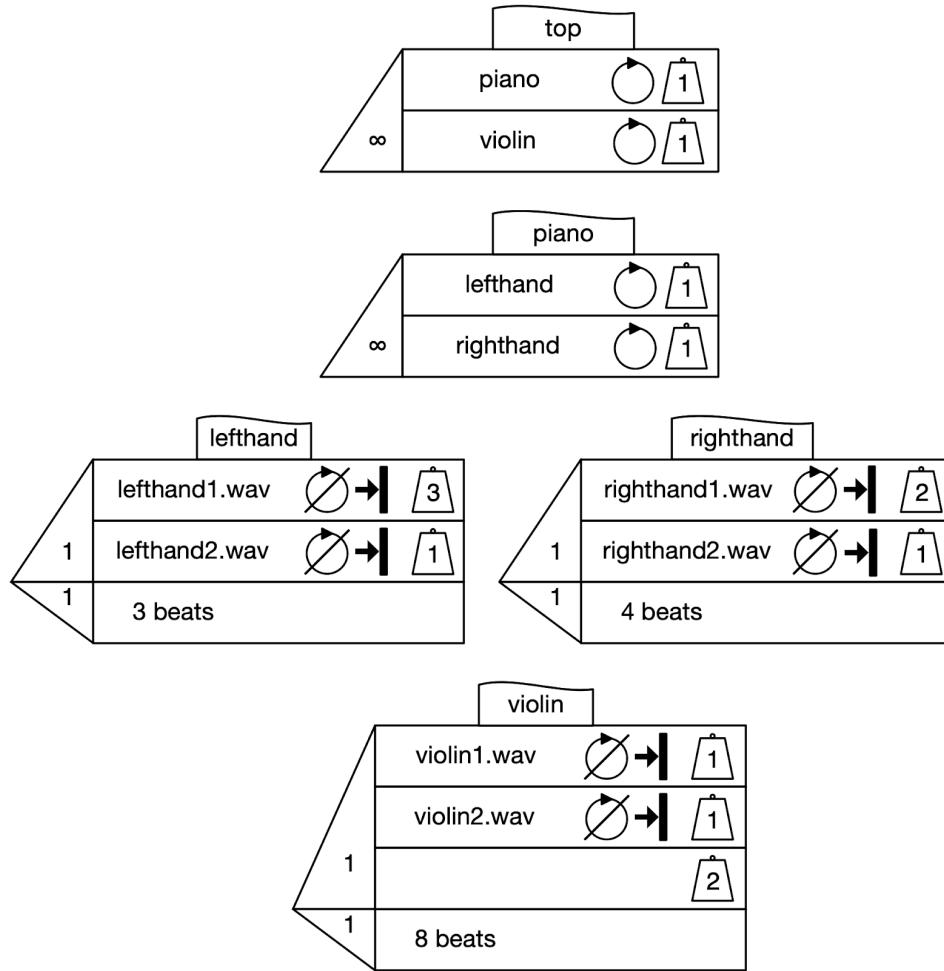


Figure 6 - The piano and violin example shown in Figure 5 with additional Time Choosers added to moderate duration. Note the use of a blank lane in the violin Chooser; if selected this lane will result in a rest (silence) for the duration of the Chooser.

As shown, Figure 6 builds on Figure 5 by adding duration controls, meaning that the user can now change these durations to make structural changes to the music. Examples of these changes include:

- Changing the ‘violin’ Time Chooser’s duration to 10 beats; if one of the eight-beat-long samples is selected this will introduce a two-beat rest before the next selection, and if the blank lane is selected there will be a 10 beat rest;
- Changing the duration of ‘lefthand’ to 4 beats; this will introduce a 1-beat rest at the end of each sample, and will align the duration with ‘righthand’. A duration of 5 beats will introduce a 2-beat rest, and so on;
- More interestingly, changing one of the ‘...hand’ Choosers to a fractional value can create phase music effects. For example, changing ‘lefthand’ to 3.5 beats creates an alternating syncopated/synchronised pattern. A duration of 3.05 beats creates a slower, tape-phase-style shift.

Example 3

The final example, shown in Figure 7, includes two doubly-nested Choosers. In order to clearly show the nesting hierarchy we present the Choosers in columns to be read from left (parent) to right (children), but the design of Choosers allows users to freely place objects as they wish. The example in Figure 7 shows beat-level durations in the child Choosers of the ‘perc’ Chooser (upper row).

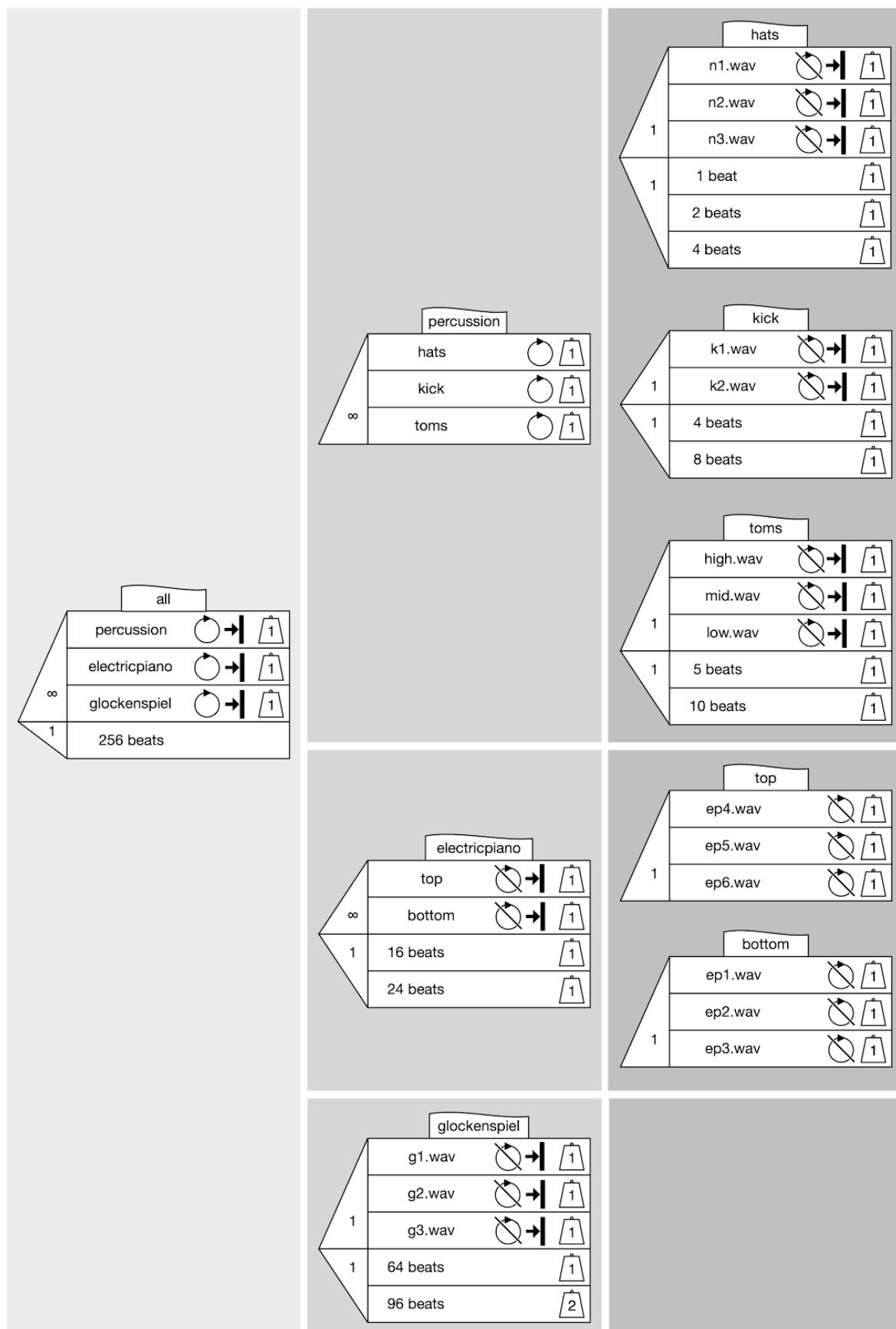


Figure 7 - An example showing doubly-nested Choosers. The nesting hierarchy has been visualised in columns and rows for the reader's benefit; in fact, Choosers and sequences may be laid out freely.

Consider the 'hats' Chooser shown in the upper right of Figure 7; when run it will select and play one of three samples without looping and, after a duration of either 1 or 2 beats, will be triggered again by the looping lane in its parent Chooser ('perc'). This, in combination with the 'kick' and 'toms' Choosers, forms the rhythmic pattern for the piece. The 'electricpiano' Chooser contains two nested Choosers, each selecting one of three electric piano samples; the 'electricpiano' Chooser will trigger new selections every 8 or 12 beats. Finally, the 'glock' Chooser is a singly-nested Chooser which will play one of three glockenspiel samples every 24 or 48 beats. The parent Chooser, 'all', has a total duration of 64 beats, after which a hard stop message will immediately stop all lanes. The structure of the piece can be changed as we have seen in the previous examples:

- Durations for each Full Chooser can be changed via Time Chooser lanes; for example, the 'hats' Chooser's durations can be shortened to create a denser, more rapidly-changing pattern. Note that the duration control for the electric piano part is in the 'electricpiano' Chooser, which triggers a selection from child Soundable Choosers ('top' and 'bottom') when it is triggered by the 'all' Chooser. Compare this to the 'perc' Chooser, in which the child Choosers ('hats', 'kick', and 'toms') control the duration of each separate musical part. Each child Chooser makes a selection of both sample and duration, plays the selected sample for the selected duration, and is then triggered again by the 'perc' Chooser's looping lanes.
- Nondeterministic choices can be controlled via Chooser nose cones and weights. Lanes can be given a zero weight to remove them from selection, and infinite weight can be added to ensure the selection of one or more lanes given a sufficient nose cone value. For example, consider the changes that could be applied to the 'all' Chooser on the far left of Figure 7; the Soundable Chooser nose cone value could be reduced to 2, meaning that two of the three lanes will be selected. The user could change the weight of 'perc' to infinity, resulting in the selection of 'perc' plus one other lane. If desired, the weight of 'glock' could be changed to 0 to remove it from selection.
- Let us now consider a specific change to the drum pattern in the upper right of Figure 7. If the user wanted to remove 'toms' from selection they could do so in one of four ways; the 'toms' lane in the 'perc' Chooser could be given a weight of zero; the Soundable Chooser nose cone of the 'toms' Chooser could be given a weight of zero (resulting in a rest of either 5 or 10 beats if the Time Chooser is not also set to zero; if both nose cones are set to zero the Chooser will be skipped completely); or the weights of all three lanes in the 'toms' Chooser could be set to zero.
- If the user wanted one glockenspiel sample to be selected and played repeatedly, rather than a new selection each time, the 'glock' lane in the 'all' Chooser could be set to not loop, and the lanes in the 'glock' Chooser set to loop.
- The 'all' Chooser has a Time Chooser with a duration of 64 beats, with all Soundable Chooser lanes set to a hard stop. This sets the total length of the piece. If the user wanted infinite playback they could change the Time Chooser nose cone to zero; set the duration to infinity; or remove the Time Chooser from 'all'.

Work-in-progress - control panel, variables

Choosers were designed to be 'tweakable'; they surface musically useful tools which allow for meaningful structural changes to be performed while minimising both viscosity and premature commitment. One musically useful example is to allow the user to change the length of a sample, and the start point for playback, via a control panel that can be opened for any sample in the system. A sketch of the control panel is shown in Figure 8.

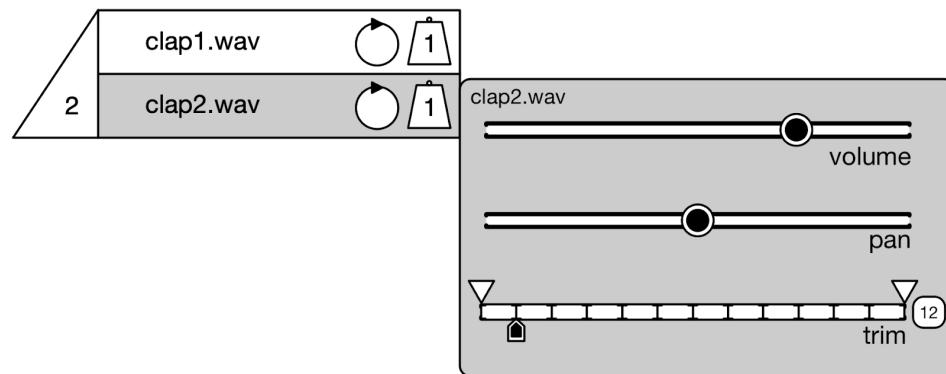


Figure 8 - A control panel to access trim (downwards arrows) and anchor point (upwards arrow).

The control panel allows the user to make a quick manual adjustment before re-running the system to audition the changes. The panel contains standard playback controls such as volume and pan. Alongside these the user can make use of a trim control, combining control over the length of the sample via ‘top’ and ‘tail’ controls (downwards-pointing arrows on the top of the horizontal line) and the starting playback position of the sample shown via an anchor control (the upwards-pointing arrow underneath the horizontal line). The user can set an optional grid, set to ‘12’ in Figure 8, which introduces a visual grid and to which the trim and anchor controls will conform. The grid can be disabled by clicking on the number, allowing freehand control of the trim and anchor settings. One notable use of this system is *Clapping Music* by Steve Reich, which can be performed making use of the Chooser in Figure 8.

An extended version of the system, beyond the scope of this paper, puts this and many other features under programmable control by end users. This extension allows variables to be assigned to numbers, samples, sequences, or choosers, and allows such variables to be used in diagrams wherever those elements might appear (e.g. trim and anchor points, nose cone values, weights, durations, and choices of sample). An interesting feature of these extensions is that simple textual expressions that alter the value of variables may appear in Soundable Chooser lanes. Such expressions are executed when the corresponding Soundable Chooser lane is selected and played, and are subject to the same nondeterministic and time-structuring controls as the soundable elements. In this way, a rich variety of nondeterministic and highly structured outcomes can be manipulated.

Conclusions

In earlier research (Bellingham et al, 2014, 2017, 2018) we identified and then implemented a number of design principles to allow non-programmers access to algorithmic composition tools. Specifically, we sought to leverage parsimony in order to enhance learnability; to surface musically meaningful actions, and to make them quick and easy; to allow both bottom-up and top-down construction; and to make use of progressive disclosure to allow for advanced use without harming usability for beginners. In this paper we have illustrated these principles in the context of exploring the range of non-deterministic algorithmic musical expression facilitated, the kinds of structural manipulations supported, and the extent to which structural malleability has been brought to the surface.

The design of Choosers allows for user preference where possible; for example, it allows for both top-down and bottom-up construction and low-viscosity changes in structure, leading to the simple graphical sequencing design shown in Figures 1 and 4. Figure 4 shows a sequence which is then populated via the named Choosers, which is an alternative to drawing a sequence arrow directly between the two Choosers. The latter would enhance the closeness of mapping; reduce hard mental operations; increase role expressivity; and reduce hidden dependencies. However, the version shown in Figure 4 presents a different set of tradeoffs; it is better suited to more complex Choosers and arrangements; viscosity is reduced and visibility enhanced; the closeness of mapping to the musical

arrangement is improved; and both juxtaposability and hard mental operations are improved when used in complex examples.

Choosers is capable of making complex music, with no limit in principle to the length of a piece, time resolution, or depth of nesting. The next phase of the project will be to test Choosers with expert users and educators, and we welcome feedback on the design thus far.

References

- Baratè, A. (2008) *Music Description and Processing: An Approach Based on Petri Nets and XML*, INTECH Open Access Publisher [Online]. Available at http://www.researchgate.net/profile/Adriano_Baratè/publication/221786820_Music_Description_and_Processing_An_Approach_Based_on_Petri_Nets_and_XML/links/004635268b80f75f6b000000.pdf.
- Bellingham, M., Holland, S. and Mulholland, P. (2014) ‘A Cognitive Dimensions analysis of interaction design for algorithmic composition software’, Boulay, B. du and Good, J. (eds), *Proceedings of Psychology of Programming Interest Group Annual Conference 2014*, University of Sussex, pp. 135–140 [Online]. Available at http://www.sussex.ac.uk/Users/bend/ppig2014/15ppig2014_submission_10.pdf.
- Bellingham, M., Holland, S. and Mulholland, P. (2017) ‘Choosers: designing a highly expressive algorithmic music composition system for non-programmers’, *2nd Conference on Computer Simulation of Musical Creativity* [Online]. Available at <http://hdl.handle.net/2436/621151>.
- Bellingham, M., Holland, S. and Mulholland, P. (2018) ‘Choosers: The design and evaluation of a visual algorithmic music composition language for non-programmers’, Church, L. and Basman, A. (eds), *Proceedings of the 29th Annual Workshop of the Psychology of Programming Interest Group - PPIG 2018* [Online]. Available at <http://www.ppig.org/sites/ppig.org/files/PPIG-2018-proceedings.pdf>.
- Bullock, J., Beattie, D. and Turner, J. (2011) ‘Integra Live: a new graphical user interface for live electronic music’, *International Conference on New Interfaces for Musical Expression* [Online]. Available at <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.449.533&rep=rep1&type=pdf>.
- Fernández, J. D. and Vico, F. (2013) ‘AI methods in algorithmic composition: A comprehensive survey’, *Journal of Artificial Intelligence Research*, pp. 513–582 [Online]. Available at <http://www.jair.org/papers/paper3908.html>.
- Green, T. R. and Petre, M. (1996) ‘Usability Analysis of Visual Programming Environments: a ‘Cognitive Dimensions’ Framework’, *Journal of Visual Languages and Computing*, vol. 7, pp. 131–174.
- Jacob, B. L. (1996) ‘Algorithmic Composition as a model of creativity’, *Organised Sound*, Cambridge University Press, vol. 1, no. 3, pp. 157–165.
- McCartney, J. (2002) ‘Rethinking the Computer Music Language: SuperCollider’, *Computer Music Journal*, vol. 26, no. 4, pp. 61–68 [Online]. Available at <http://dx.doi.org/10.1162/014892602320991383>.
- Puckette, M. (1991) ‘Combining Event and Signal Processing in the MAX Graphical Programming Environment’, *Computer Music Journal*, The MIT Press, vol. 15, no. 3, pp. 68–77 [Online]. Available at <http://www.jstor.org/stable/3680767>.
- Wilson, S., Cottle, D. and Collins, N. (2011) *The SuperCollider Book*, MIT Press.

The Naturalist's Friend

A case study and blueprint for pluralist data tools and infrastructure

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

Abstract

Spreadsheets liberated individual end users to own their own data and curate its structure and relationships. However, most realistic individuals are embedded in multiple overlapping communities and societies, each of which brings different visions, ontologies and working practices. No modern digital tools perform the same liberating functions for communities that spreadsheets did for individuals. We will sketch the working patterns and relationships amongst some communities of naturalists that we have studied, distil these down into the description of a set of “challenge problems” and then sketch the design and infrastructure of a tool, codenamed “The Naturalist’s Friend”. This tool will enable multiple communities to collaborate on simple, tabular-structured data for which they have mismatched criteria for membership and interpretation of both rows and columns, without obliging any of them to compromise their autonomy over their own data standards, or lose the modern digital affordances of constantly live access to version-managed and publicly addressable incarnations of their shared data.

1. Introduction

Modern digital tools have an innate centralising, authoritarian tendency. Communities using them are obliged to converge on shared, central ontologies and workflows, or else be inflicted with inefficient and archaic data sharing practices such as emailing spreadsheets to one other. As communities grow larger and longer-lived, these archaic workflows eventually impose huge costs as it becomes impossible to determine what is the most recent, authoritative version of a resource for a community without costly and error-prone manual checking. The costs of filing and locating this data swamp all but the smallest communities operating on the simplest data.

1.1. Case Studies

We will present case studies of communities of naturalists cooperating to organise their data on observations and identifications of organisms, building on previous such studies in (Basman & Tchernavskij, 2018; Tchernavskij, Basman, Nouwens, & Beaudouin-Lafon, 2019). This area of data work is attractive because the absence of a direct profit motive enables economic concerns to be disentangled to some extent from commercial ones, and further because these communities already have centuries of experience in working with constantly shifting and constantly negotiated plural ontologies.

Following (Tchernavskij et al., 2019) we will cite larger-scale examples of extended naturalist communities trying to share more substantial sets of data (tens of individuals sharing thousands of species and observations) but also gain perspective by supplementing these with recent experiences in working on similar problems at a much smaller scale (two individuals sharing hundreds of records).

1.2. Challenge Problems

After tracing these sample communities, we will distil a collection of typical collaboration patterns into a selection of “challenge problems”. In order to focus these problems and make the design space tractable, we will concentrate on the simplest practical data substrates, rectangular grids with a regular row structure, that space well-described by the ubiquitous .CSV files.

These substrates will prove quite adequately rich in intractable problems. Those which are more tractable relate to shared ownership over rows — communities may have different, and shifting standards for inclusion of records of relevance.

Less tractable are those which relate to the meaning and relations amongst columns. For example, almost every community of naturalists will include a column identifying the species of the observation. However, different communities will defer to different authorities that they recognise as competent to

assign such names, and even those which agree on a name may well disagree on the taxonomical interpretation of the name. Furthermore, the set of authorities relevant for a particular community will shift over time — in the face of such mismatches, all collaborating communities will want to stay in continuous contact, sharing access to the records of shared interest without loss of data or loss of meaning of data.

2. Small-Scale Case Study

In this section, we present a very small-scale example of data sharing, involving only 2 participants and 4 documents. The situation of collaboration focused on an upcoming visit by the two participants to Groton Wood in Suffolk, well-visited over 40 years by naturalist Oliver Rackham, whose visits had been carefully noted in 4 notebooks amongst the digitised archive held by Cambridge University Library¹. The two participants were A, an experienced technologist with a weak amateur naturalist knowledge, and B, an experienced ecologist with good knowledge of the standard office tools (Excel, Word, etc.) that his profession involved. The purpose of the visit was to apply Rackham’s observations to guide an appreciation of the wood’s habitat, determine which of the species noted over the years by Rackham could still be discovered in the wood, and perhaps note a few more.

2.1. Initial Compilation

Participant A devoted some time to deciphering the handwriting in the notebooks, and trying to compile an exhaustive index of each particular visit’s sightings. This produced the first document, a very wide spreadsheet with 61 columns (as well as the observation data, one column for (nearly) each visit by Rackham to Groton) and 206 rows, one for each observed species. A selection of a later revision of this document, named “Document A1₃”, is shown in Figure 1. In order to compile these data, A attempted to compensate for his lack of botanical knowledge by exploiting various online resources, especially the autocomplete facilities of websites such as BSBI² and iNaturalist³.

As it transpired some weeks later, participant B had also compiled his own list, which was then emailed to A in the form of an Excel spreadsheet, a selection of which, named “Document B1₁”, is shown in Figure 3. This spreadsheet has 3 columns and 146 rows. This immediately presented a somewhat interesting data normalisation challenge. Whilst the “scientific name” field should have been expected to be in common between the two documents, in practice there are quite some subtleties in this area. Whilst A had believed in adopting the BSBI’s name (B’s suggested online resource) for a taxon whenever it conflicted with iNaturalist’s, they would end up with a result more reflective of accepted scientific research, it soon transpired that B’s taxa were drawn from yet another source, (Stace & Thompson, 2019)⁴. Being the most recent version of the resource accepted by UK professionals, its contents were naturally accepted as authoritative for our application. However, given the further applications we desired for the data, it was necessary to maintain the linkage to the digital resources — this is the reason that A’s larger spreadsheet shown in Figure 1 includes URLs for both of the online resources, and in a couple of cases the three resources referenced for a species each refer to it by a different name.

¹available at <https://cudl.lib.cam.ac.uk/collections/rackham/1>

²The Botanical Society of Britain & Ireland, whose Online Atlas of the British and Irish Flora is available at <https://www.brc.ac.uk/plantatlas/> — this was already a resource suggested by participant B

³Primarily a citizen science platform, already described in (Basman & Tchernavskij, 2018; Tchernavskij et al., 2019), whose taxon interface is available at <https://www.inaturalist.org/taxa/>

⁴This a bulky and expensive paper reference, very recently published, whose information at the time of writing is still not widely available in digital form, and which is now even temporarily out of print due to its extreme popularity.

	A	B	C	D	E	F	G	H	I	BC	BD	BE	BF	BG	BH	BI
	File	Edit	View	Insert	Format	Data	Tools	Add-ons	Help	All changes saved in Drive						
1	Ordinal	Rachian's notation	Species/Genus	Common name	Nipplewort	BRC iNK	DV									
2	1 [Lapsum?]		Lapsana communis	Common Name	https://www.naturalist.org/navai/55981-Lapsana-communis	https://www.brc.ac.uk/plantatlas.c	8	1								1
3	2 Cirs vulg		Cirsium vulgare	Common Name	https://www.naturalist.org/navai/2289-Cirsium-vulgare	https://www.brc.ac.uk/plantatlas.c	11	1								1
4	3 Chamaener		Chamaenerion angustifolium	Common Name	https://www.naturalist.org/navai/56169-Chamaenerion-angustifolium	https://www.brc.ac.uk/plantatlas.c	20	1								1
5	4 Scroph nod		Scrophularia nodosa	Common Name	https://www.naturalist.org/navai/12445-Scrophularia-nodosa	https://www.brc.ac.uk/plantatlas.c	0.5	12	1							1
6	5 [Geum viv?]		Geum rivale	Water Avens	https://www.naturalist.org/navai/55159-Geum-rivale	https://www.brc.ac.uk/plantatlas.c	1	1								
7	6 Ranun rep		Ranunculus repens	Creeping buttercup	https://www.naturalist.org/navai/48229-Ranunculus-repens	https://www.brc.ac.uk/plantatlas.c	5	1								
8	7 Dryop d affnum		Dryopteris affinis	Scaly Male Fern	https://www.naturalist.org/navai/321288-Dryopteris-affinis	https://www.brc.ac.uk/plantatlas.c	1	1								
9	8 Arctum		Arctium lappa	Greater/Lesser Burdock	https://www.naturalist.org/navai/5501-Arctium-lappa	https://www.brc.ac.uk/plantatlas.c	15.1-3s									1
10	9 [Pulen ster?]		Potentilla sterilis	Barein Strawberry	https://www.naturalist.org/navai/8855-Potentilla-sterilis	https://www.brc.ac.uk/plantatlas.c	1	9								1
11	10 Senec eruc		Jacobaea erucifolia	Horn Ragwort	https://www.naturalist.org/navai/168817-Senecio-erucifolia	https://www.brc.ac.uk/plantatlas.c	9	1								1
12	11 Dactylis		Dactylis glomerata	Cock's-Foot	https://www.naturalist.org/navai/52229-Dactylis-glomerata	https://www.brc.ac.uk/plantatlas.c	4	1								
13	12 [Gulmenc?]		Hypericum perforatum	Perforate St. John's Wort	https://www.naturalist.org/navai/56077-Hypericum-perforatum	https://www.brc.ac.uk/plantatlas.c	1	5	0							
14	13 Hyp pfor															

Figure 1 – Participant A’s data collection (in final form) - Document A1₃

	A	B	C	D	E	F	G	H	I	BC	BD	BE	BF	BG	BH	BI
	File	Edit	View	Insert	Format	Data	Tools	Add-ons	Help	All changes saved in Drive						
1	Ordinal	Rachian's notation	Species/Genus	Common name	Nipplewort	BRC iNK	DV									
2																
3																
4																
5	5 [Geum viv?]		Geum rivale	Water Avens	https://www.naturalist.org/navai/55159-Geum-rivale	https://www.brc.ac.uk/plantatlas.c	1	1								
79	Ranunc auricomus		Ranunculus auricomus	Goldilocks Buttercup	https://www.naturalist.org/navai/56077-Hypericum-perforatum	https://www.brc.ac.uk/plantatlas.c	1	1								

Figure 2 – Participant A’s “illustrated guide of highlights for amateurs” - Document A2



A	B	C	D	E	F	G	H	I
1 Scientific Name	Common Name	AWI						
2 Acer campestre	Field Maple	x						
3 Agrimonia eupatoria	Agrimony							
4 Ajuga reptans	Bugle							
5 Alisma plantago-aquatica	Water Plantain							
6 Anemone nemorosa	Wood anemone	x						
7 Angelica sylvestris	Angelica							
8 Arctium minus	Lesser Burdock							
9 Artemisia vulgaris	Mugwort							
10 Arum maculatum	Lords and Ladies							
11 Athyrium filix-femina?	Lady-fern							
12 Azolla filiculoides	Water Fern							
13 Betula	Birch							
14 Bryonia dioica	White Bryony							
15 Callitricha sp.	Starwort							
16 Cardamine flexuosa	Wavy Bitter-cress							
17 Cardamine pratensis	Cuckoo Flower							
18 Carex pallescens	Pale Sedge	x						
19 Carex pendula	Pendulous Sedge	x						
20 Carex pseudocyperus	Cyperus Sedge							
21 Carex remota	Remote Sedge	x						
22 Carex sylvatica	Wood Sedge	x						
23 Carpinus betulus?	Hornbeam	x						
24 Centaurea nigra?	Common Knapweed							
25 Chamaenerion angustifolium	Rosebay Willowherb							
26 Circaea lutetiana	Enchanter's-nightshade							

Figure 3 – Participant B’s original data collection - Document B1₁

A	B	C	D	E	F	G	H	I	J	K	L	M
1 Scientific Name	Common Name	AWI	In AB	AB notes	DV notes							
2 Acer campestre	Field Maple	x	T									
3 Agrimonia eupatoria	Agrimony			1								
4 Ajuga reptans	Bugle			1								
5 Alisma plantago-aquatica	Water Plantain			1								
6 Anemone nemorosa	Wood Anemone	x		1								
7 Angelica sylvestris	Angelica			1								
8 Arctium minus	Lesser Burdock			1 I chose Arctium lappa - check A. minus is generally more common than A. lappa but there is a chance it could be								
9 Artemisia vulgaris	Mugwort			1								
10 Arum maculatum	Lords and Ladies			1								
11 Athyrium filix-femina?	Lady-fern			1 As genus - chosen, 1978								

Figure 4 – A’s importation of B’s B1₁ as AB1 for harmonisation (final version AB1₃)

2.2. Rounds of Harmonisation

A did the work to normalise B’s authoritative names into column B of his spreadsheet which then made possible the corresponding normalisation of determining which species he and B actually had in common — in practice, both transcribers had missed different sets of Rackham’s observations (which also in themselves sometimes diverged in species name from all the authorities in play). This normalisation had to be done by time-honoured, manual techniques — A’s spreadsheet A1 was enhanced with a boolean column “in DV” and B’s spreadsheet B₁ was imported by A as AB1 (Figure 4) with a new column “in AB”, each was sorted into alphabetical order and the checkboxes in the opposing spreadsheet were checked off. At this point, further discrepancies between A and B’s ontologies for the problem surfaced. In A’s transcription, as a result of his domain inexperience, two large categories of species had been systematically omitted — i) all trees, ii) all extremely common species with which he was familiar, e.g. nettles, brambles and the like. This was on the basis that such species would be both so frequently mentioned in the notes, and frequently encountered in the field, that observing them would have little informational value. On the other hand, B’s professional experience had trained him that no such observation should be considered beneath notice. However, on the other side, A had also transcribed all fungi from Rackham’s notebooks, which B had omitted since i) this was an area outside his professional expertise, and ii) since the proposed visit was in spring, few of these would be visible. On the issue of the trees, A decided for the while to persist in his stubbornness and constructed an extra annotation T in his working copy of B’s spreadsheet in order to encode that a species was missing in his own for this reason. A then projected his excess rows into B’s schema and shared with B via Google Sheets an enhanced version of their original spreadsheet in their schema, with the additional observations filled in. In practice this led to two elaborated resources, A1₂ and AB1₂, whose columns were supersets of those present in the original A1₁ and AB1₂. A then went through a further round of transcription from the

original notebooks in order to turn up not only all of B's excess observations, but a couple further that had been missed by both transcribers, resulting in *A1₃* and *AB1₃*.

2.3. Compilation of Derived Resources

In this section we now describe two further resources that were derived from the basic transcriptions *A1* and *B1*.

2.3.1. Visual Checklist for Amateurs

At this point it was time to use the pooled information to derive two further kinds of artefact, primarily to be used in the field at the actual visit. The first kind was an illustrated guide to a couple of dozen of the most frequently observed "Ancient Woodland Indicators" (AWIs)⁵, since as well as A, three and a half other amateur naturalists would be accompanying on the trip, to whom the species would be largely new. A selection of the available images for the taxon available on iNaturalist via column E of resource *A1* was examined, and A selected two which in his opinion gave an impression both of the overall habit of the plant and then some detail which he found interesting. This gave rise to a derived resource *A2* (shown in Figure 2 by A from document *A1*, which corresponded to only a small subset of the rows from *A1* - in this case, to those which had been marked as AWIs, and fell into the top 25 of species when sorted by number of mentions in Rackham's notebooks.

This process of image selection is an interesting one to which we will return in the next section, but it is well worth noting that the level of experience of the transcriber will have a significant bearing on the choice and value of the resulting selection. An inexperienced transcriber may well end up selecting an image which is visually appealing, but is useless for resolving some of the significant details which would have to be used to make an reliable identification of the species in the field. On the other hand, an expert botanist might well end up selecting images which were less visually appealing but showed significant details, but on a yet further hand, an expert botanist may well not compile such a visual list at all since experts tend to derive their identifications from written descriptions and professionally compiled dichotomous keys, and this checklist *A2₁* itself was indeed primarily aimed at the needs of amateurs. However, this highlights that such a visual checklist is a form of "community resource" and a particular selection may well end up serving the needs of some communities of interest better than others.

2.3.2. Full Checklist for Observations

A and B on their actual visit to Groton would need an uncluttered template document in which to enter their observations in the field. This would include the bare minimum information from the transcription documents *A1*, *B1* of the (or a) scientific name for the species, its common name, AWI status, and then room in which to write notes of the observation. A drew up his own version of this resource, named *A3* which is shown in Figure 5.

B in the meantime drew up their own rendition of the checklist for his own use, which is based on essentially the same data but with fewer columns, named *B3* shown in Figure 6. It's worth noting that whilst by this point the discrepancy between the two checklists with respect to inclusion of trees had been resolved, that B once got to the field expressed disappointment that somehow the common species that had been in their original collection *B1₁* had gone astray from their *B3*. This illustrates the huge potential for accidental loss or corruption of data as it passes through clunky, manual workflows such as these since each instance of the data is only as good as the last visual check made to correlate it with its informationally disconnected sources.

2.4. Information Flow Between Documents

The complete information flow network between the different documents in this collaboration is illustrated in figure 7. It would clutter the diagram unduly to try to visually represent the detailed nature of the data relationships, but the previous discussion should make clear that as well as straightforward cases where extra columns or rows are introduced into upstream resources, there are several instances of

⁵Plants whose presence which, on account of their extremely slow colonisation of adjacent forest, marked the area they were found in as "ancient woodland", that is, likely to have been continuously occupied by woodland for upwards of 400 years.

Checklist of Groton Wood spp from Rackham's Notebooks						
A	B	C	D	E	F	G
1	Common Species/Common Name	AB1	ORM	DR	Qual	Time
+ A = Abundant, B = Common, C = Infrequent, D = Occasional, E = Rare, F = Scarce, G = Uncommon						
+ Trees and shrubs						
+ * = Species first mentioned in Oliver Rackham's notebooks						
D% = GAF% (relative abundance) D=Dominant, A=Abundant, F=Infrequent, O=Occasional, R=Rare						
Qual = Qualifiers T=tame Tree, Y=young Tree, S=Seedling or Sapling, C=Clumped, F=Patchy, W=Wet, P=Possibly planted, ?=D=uncertain						
3	T1 Acer campestre+	Field Maple	1			
4	199 Adoxa moschatellina	Moschatel	1	1		
5	61 Agrimonia eupatoria	Common Agrimony	1			
6	198 Agrostis stolonifera	Creeping Bent	2			
7	163 Ajuga reptans	Bugle	12			
8	88 Alisma plantago-aquatica	Water-Pantain	5			
9	196 Alnus cordata pratinoides	Meadow Sweet	1			
10	173 Anemone nemorosa	Wood Anemone	14			
11	191 Angelica sylvestris	Wild Angelica	1			
12	8 Arctium lappa (minus)	(Greater/Lesser) Burdock	15			
13	161 Artemisia vulgaris	Common Mugwort	1			
14	204 Arum maculatum	Lords and Ladies	1			
15	86 Athyrium filix-femina	Lady Fern	2			
16	126 Azolla filiculoides	Water Fern	4			
17	T2 Betula+	Birch				
18	201 Brachypodium sylvaticum	Slender False Brome	1			
19	46 Bryonia cretica (diosca)	White Bryony	3			
20	31 Callitricha	Water Starwort	6			

Figure 5 – Participant A’s checklist for the field - Resource A3

A	B	C	D	E	F
1 Species/Genus	Common name	ORM	DR	Qual	Notes
2 Acer campestre+	Field Maple	1			
3 Adoxa moschatellina	Moschatel	1			
4 Agrimonia eupatoria	Common Agrimony	1			
5 Agrostis stolonifera	Creeping Bent	2			
6 Ajuga reptans	Bugle	12			
7 Alisma plantago-aquatica	Water-Pantain	5			
8 Alnus cordata pratinoides	Meadow Sweet	1			
9 Anemone nemorosa	Wood Anemone	14			
10 Angelica sylvestris	Wild Angelica	1			
11 Achium (leptophyllum)	(Greater/Lesser) Burdock	15			
12 Artemisia vulgaris	Common Mugwort	1			
13 Arum maculatum	Lords and Ladies	1			
14 Athyrium filix-femina	Lady Fern	2			
15 Azolla filiculoides	Water Fern	4			
16 Betula+	Birch				
17 Brachypodium sylvaticum	Slender False Brome	1			
18 Bryonia dioica	White Bryony	3			
19 Callitricha	Water Starwort	6			
20 Cardamine flexuosa	Wavy Buttercress	4			
21 Cardamine pratensis	Cuckooflower	10			
22 Carex oederiana	False Fox-sedge	1			
23 Carex pallescens	Pale Sedge	1			
24 Carex pendula*	Pendulous Sedge	1			
25 Carex pseudocyperus	Cyperus Sedge	23			
26 Carex remota	Remote Sedge	12			
27 Carex spicata	Spiked Sedge	1			
28 Carex sylvatica	Wood Sedge	13			
29 Carpinus betulus?*	Hornbeam	1			
30 Centaurea or Centaura	Centaurea or knapweed	4			
31 Ceratium fontanum	Common Mouse-ear	2			

Figure 6 – Participant B’s checklist for the field - Resource B3

complex multilateral relationships. An example of such a relationship is between AB₁₃ and its parents A₁₃ and AB₁₂ (and ultimately B₁₁), where neither the rows nor columns of these documents are proper subsets of either of their parents — along each axis, some material has been “inherited” from one parent, then possibly some of that information has been selected away some policy, and then enlarged by a subset taken from another parent. It is possible that a more or less regular description of this policy could be expressed in some declarative form, perhaps drawn from the domain of “lenses” promoted by the bidirectional programming framework of (Foster & Pierce, 2009). However, we should, in the light of this particular process, note some further subtleties.

Even in our sample community of 2 participants, we find that the issues of authority and relevance are central. As the domain expert, any of B’s data decisions should expect to take priority over any made by A, and one would expect the natural structure of data updates within this network to reflect this. In fact, that B’s updates took the form of emailed spreadsheets ensured that no data that he “owned” might get modified by A, except inadvertently. One would expect that any technological solution to such collaboration problems would be able to make it inherently clear to all participants what the nature of such authority relations might be — for example, in this situation both A and B would both be mutually, implicitly clear that A would perform no actions that might cause data notionally “owned” by B to be updated, unless B had explicitly requested or confirmed this.

One suspects that the difficulty in reflecting such complex, and possibly shifting relations of ownership and authority over different subsets of the data is what drives most collaborators back to the “clunk workflow” of emailing spreadsheets, since at least in that model it is clear what data has been transferred and no possibility of it shifting under the participants’ feet. However, even in this small-scale collaboration the lack of technological support led to considerable inefficiencies and numerous errors, many of which have probably not even yet been characterised.

This collaboration is an endless story of “broken relations” — each of the 5 documents in their various incarnations morally is connected to the others via what should be relations, and should any of them become updated, one would hope it would be possible to choose through some automated means to propagate the corresponding updates to the others. For example, should a fresh decision be made about the accepted species name for an element of B₁, that this could be propagated by some automated process to A₁, A₂ A₃ and B₃. Right now this update would have to be copied by hand between the involved documents.

However, it would be obviously inappropriate for these updates always to happen transparently and automatically, as it would if Figure 7 represented a conventional kind of “dataflow network”. Each author has their own interest in the integrity and meaning of their documents, and whilst in some cases

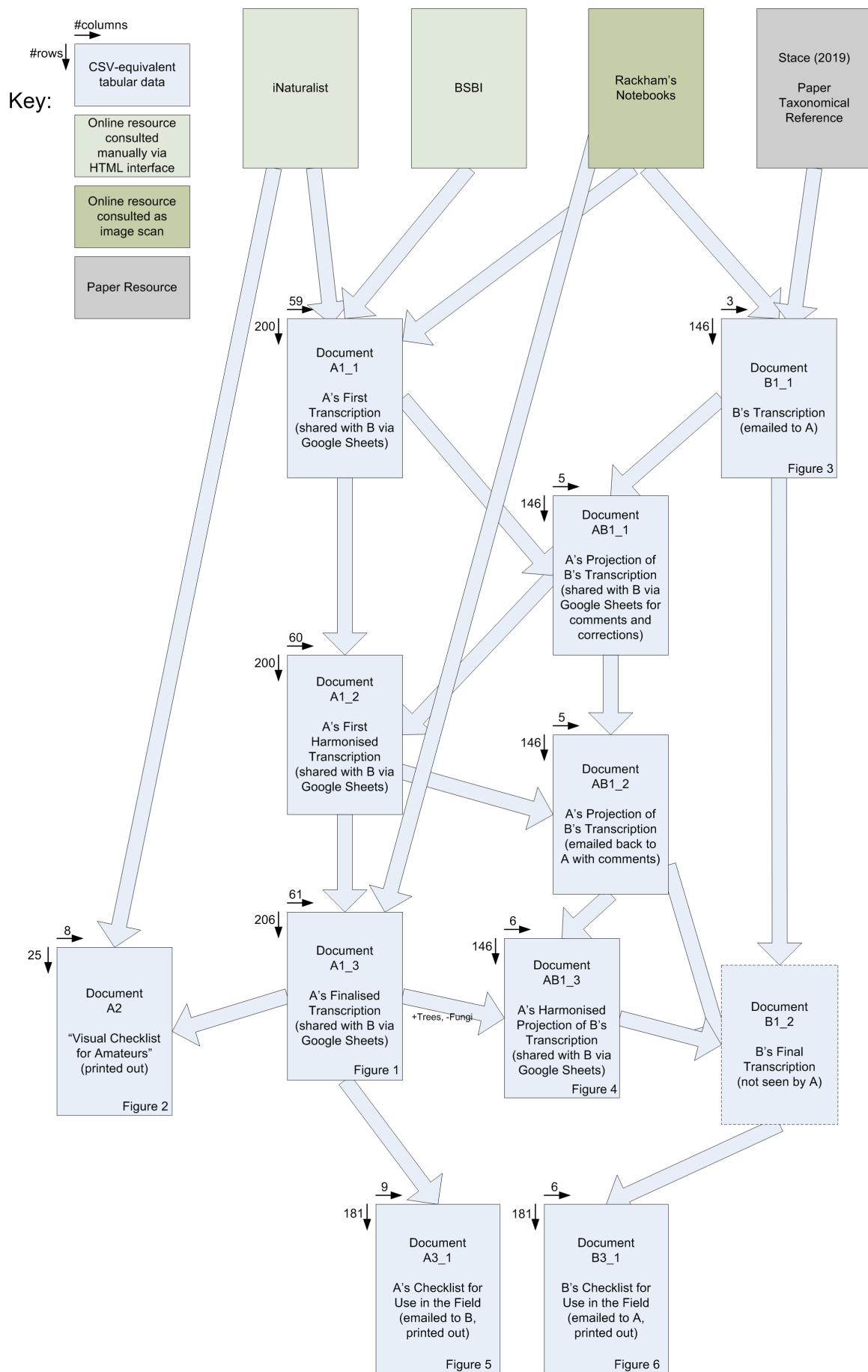


Figure 7 – Complete information flow between documents in small-scale collaboration

they might want to set a policy of automatically accepting certain kinds of updates, they would want to preview them and perhaps temporarily or permanently reject them.

3. Larger-Scale Case Study

In this section we will consider the case of a larger group of collaborators, a increasingly connected group of naturalists working in the Salish Sea, a network of waterways stretching between British Columbia and Washington State and encompassing Vancouver Island and several neighbouring islands. Primarily these groups are composed of professional naturalists, collecting much larger datasets than those in section 2, typically of thousands or tens of thousands of observations over a timescale of years or decades. Participants from these groups consult a variety of specialised online and paper resources to derive taxonomies and checklists of species, and collect observations through a variety of methodologies recorded in a variety of formats. Some of the goals of the growing collaboration are to produce tools where some core of the community data may be rendered into some common forms in order to facilitate statistical and visual studies of species occurrence across the region as a whole, without forcing the communities to compromise on local imperatives for data collection and representation. A representation of some of the communities involved and some data that they have shared, together with some upstream information sources is shown in Figure 8.

Given the size and number of the communities we are representing here, Figure 8 suppresses numerous details that would be visible had we tried to draw a diagram of detailed interactions in these communities at the same scale we have shown in Figure 7. In conversations with the members of these communities, both those pursued for (Tchernavskij et al., 2019) and privately, we have seen that on each occasion these communities interact over their shared data, tangles such as that shown in 7 and much worse will routinely arise.

3.1. An ecology of emailed spreadsheets

The working practices shown in 2 are found to be typical in this much larger, more long-lived community, only the costs imposed are more severe. Spreadsheets with thousands of entries are routinely mailed back and forth between participants, all with divergent schemas and information contents, and referred to discrepant taxonomies. The task of normalising one of these documents with respect to the information standards of a different community may be a long-boiling task of several weeks rather than just an evening annoyance. One of the participants has explained that their community is sitting on a file server holding decades worth of such documents whose contents may never again be understood or indexed unless they are lucky enough to experience the transient tenure of a graduate student with the relevant skills who might succeed in mining a handful of them. Since we've given the flavour of such problems in section 2 we will draw a veil over the details except to talk a little further of an domain issue which is exposed more sharply in this larger-scaled community, that of taxonomies.

3.2. Taxonomies in the large

Given these are peer communities of experts, the issue of taxonomical choices becomes more substantial than it was during our small-scale collaboration, and exposes some of the core requirements on our "Naturalist's Friend" application that we will try to sketch out in section 4, that go beyond those of data tools so far developed. Figure 8 shows some of the many resources that our communities might consult for taxonomical information, and not shown are many further resources that might exist only on paper, or even in the form of personal contact with individual subject-matter experts who might answer questions from their own personal experience. Whilst there is obvious value in centralising as much shared knowledge as possible in a centralised "taxon resolution database" (TRD) as shown in the centre of the Figure, it's clear that, as with, for example, iNaturalist's taxon database itself, no single choice is going to be wholly valid for all participants. The level of databases highlighted with question marks in the row below the central TRD suggests that each community member would in theory expect to administer choices about a TRD local just to their project. A particularly salient example of this is the Metchosin project, which, given it collaborates particularly often with federal authorities publishing lists of endangered species drawn from a nationally standardised taxonomy, needs to be able to refer records

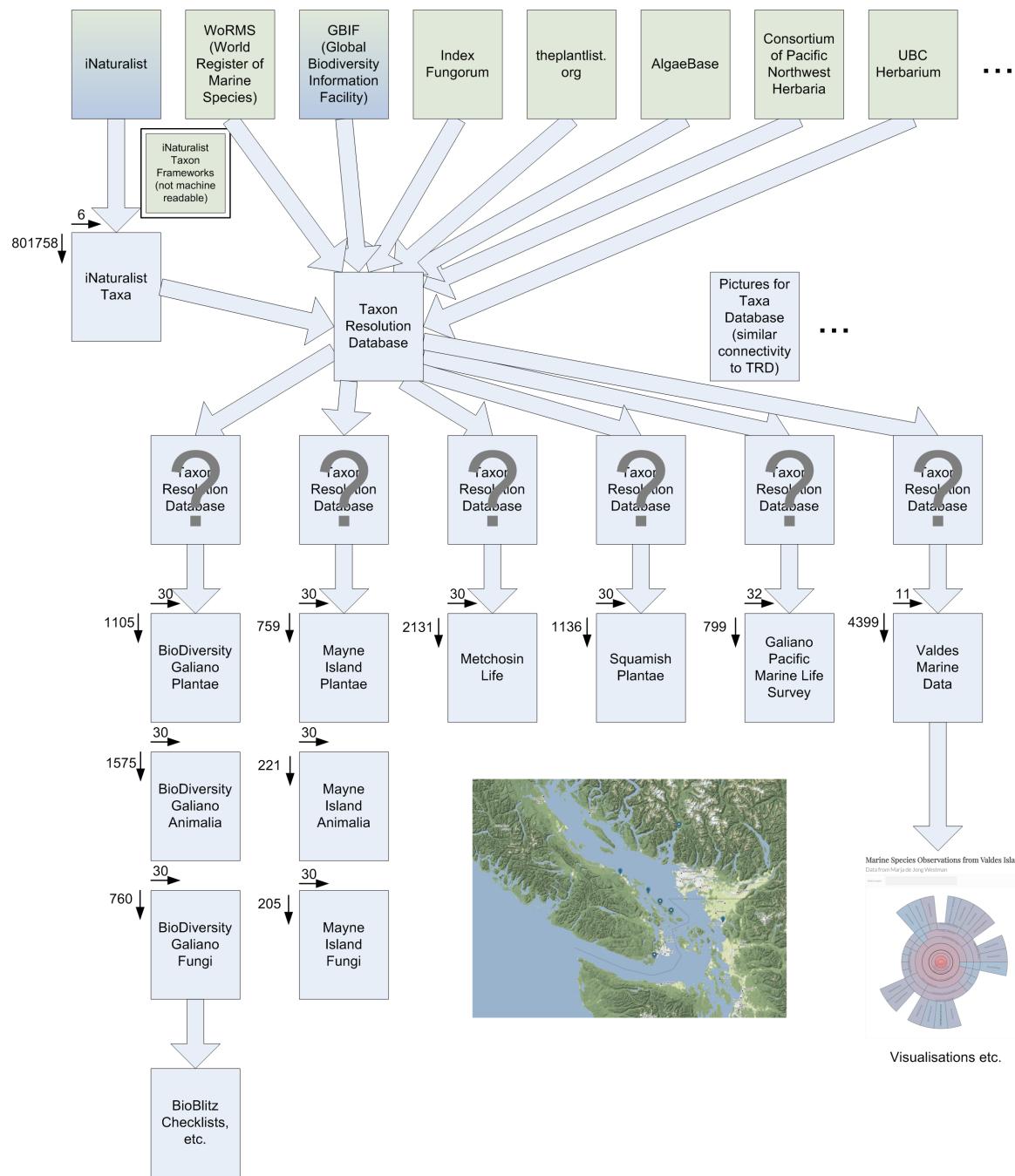


Figure 8 – Partial information flow between documents in large-scale Salish Sea collaboration

to that taxonomy. But in practice, it would be impossible for any participant to completely administer a local taxonomy and express design choices, for example, about a list of 800,000 species — instead, they will want to issue strategic directives at a high level of granularity, e.g. “For all organisms within this taxon we defer to authority A’s published list, and within this other taxon we defer to authority B with the exception of the following 5 species for which we override their choice”.

This is in fact a problem which the iNaturalist platform itself contains facilities for managing, in its “Taxon Frameworks” system⁶. Unfortunately this system has a few drawbacks. Firstly, it is implemented only with respect to iNaturalist’s taxonomy itself, and its encoding of the taxonomic relationships is not made public in a machine-readable form to external users (represented in Figure 8 by the black box drawn around its document). The Taxon Frameworks system can’t be used by 3rd parties to administer taxonomies for their own communities, without forking and hosting an entire iNaturalist instance devoted to their community. This is prohibitive for reasons discussed in (Basman & Tchernavskij, 2018; Tchernavskij et al., 2019). Secondly, the Taxon Frameworks UI is rather intricate, a little clunky since lightly used by most iNaturalist users, and aimed towards experts who are thinly represented in their userbase. Nonetheless, the functionality it offers is a useful model for affordances we would want to democratise. Useful HTML renderings of its taxonomic mappings and their relationships to upstream authorities are available at URLs such as https://www.inaturalist.org/taxa/1/taxonomy_details.

4. Design Sketch

In this section we will try to sketch out the parameters of our proposed application and data infrastructure, codenamed “The Naturalist’s Friend”, aimed at mitigating the kinds of problems we’ve described in sections 2 and 3.

The interface of our desired tool will physically resemble as closely as possible that of one of the widely deployed spreadsheet tools (Office, Numbers, Google Sheets, etc.) in order to allow users to leverage the skills and experience they’ve gained in these end user tools. We will simplify our initial design to a great extent by removing the facility for any formulae or traditional “computed values” since these are not of particular relevance to our communities of interest. However, instead, we will augment the interface as a whole by extra controls which allow the community to make and break relations to other similar data — either data about related but distinct items, such as the taxonomy of observations, or data about the same class of item, such as observations made by other communities. These relations will always be “forgiving” in that violations of the relation will not be treated as an error condition by the interface, but just as a further feature of the data environment which the user may choose to highlight. Since in addition there will be ready access to high-integrity snapshots of past versions of the data as well as its accompanying relations, the effect of accepting or rejecting potentially destructive bulk updates of the data induced by data can always be undone as easily as they are made.

However, following (Basman & Tchernavskij, 2018), it will be impossible to specify the interface of our tool in separation from its infrastructure and externalised affordances — since it is a crucial part of its specification that it must work for its communities “in place”. For example, at any time a user should have instant access to the basic substrate of their data — in this case, a .CSV file at an easily determinable URL or direct download. Further, there should be similarly directly externalised access to the version history of the community’s own data in a publicly intelligible format — we recommend a standard git repository, and furthermore to mirror community accounts and relations by corresponding ones within github⁷. In addition, the packaging and distribution of the data itself will also be geared through a widespread repository system such as npm, where the data is augmented with a simply-structured JSON metadata block in a format perhaps derived from that of the Frictionless Data project from Open Knowledge Labs⁸.

⁶Described at https://www.inaturalist.org/pages/taxon_frameworks

⁷This is not an irreversible design choice — communities should be able to host their own storage and versioning apparatus, but a useful default would assign them to these, the most widely deployed technology choices for this class of problem

⁸<http://okfnlabs.org/projects/frictionless-data/>

A crucial community-oriented principle of design is that the tool must pay its way even in a community where the individual user is its only adopter. This implies that it must behave no worse than a traditional spreadsheet in any aspect that is relevant for its use in the planned communities, and hopefully it will behave better in many aspects. This implies that the traditional archaic workflow where an incoming collaboration takes the form of a spreadsheet emailed to the user must be a centrally designed use case. Another implication of this principle is the approach to “forgiving relations” mentioned in an earlier paragraph — the presence of a relational constraint in the system should not make it less usable for the purposes of plain, literal data entry than it was before.

With this, we end up with an interface that resembles, in one pane, a highly simplified spreadsheet application, which is only capable of working one CSV-equivalent literal data, but which presents another pane, the “community pane” showing a view structurally similar to that in Figure 7, showing the relationships of the currently focused data to related documents in the community, some of which will be documents in different schemas managed by this or other users, and some of which will be significant different versions of the current document. The notion of “significance” will typically be that two community members have interacted by exchanging or signing off the document version for some purpose.

4.1. “Make Relation” and “Pull Relation” actions

The interface will support some interesting operations not typically seen in office applications. The most important of these is the “make relation” operation. This will, given another resource either stored on disk or available on the network, instruct the system that it bears some relation to the current document. This relation will usually take the form of selecting some subsets of columns in the corresponding documents, and selecting from a set of available “lenses” which map the values one onto the other. An example of such a lens might be a “taxonomical lens” which maps the names of species used in one community to those used in another, with reference to one of the taxon resolution databases described in section 3.2. Another example might be to map geographical coordinates expressed in decimal latitude and longitude as used by modern cloud-based maps such a Google Maps into UK Ordnance Survey coordinates which have been used by historical and professional communities.

Along with defining such lenses for values, the interface will allow the user to select or define a mapping which explains the presence or absence of rows in the document based on functions of the column values — e.g. “The version of this data held by community B does not include any of our rows for which the value of column *kingdom* is *Fungi*. An instance of such a relation is the one connecting documents *A1₃* and *B1₃* seen in section 2. A more ambitious example would be to ask the system to synthesize a first pass at document *AB₁* given documents *A1₁* and *B1₁* — in this case, the schemas of the parent documents are quite different, and the taxonomies are also misaligned.

It’s an overriding design principle that these relations/lenses will not be hard relational constraints such as those seen in relational databases, or more recently in the “linked data” initiative described at <http://linkeddata.org/>, but soft constraints that the only consequence of violating will be that the data not respecting the constraint will be highlighted in some views — the “forgiving relations” described in the opening of this section. However, whenever an author is happy with the consequences, or merely as an experiment, they may choose the “Pull Relation” action for such a lens, which will update their own document with the corresponding mapped values. Given every document in the system is durably version managed, these experiments can be reversed as quickly as they are made. An example of this might be for the owner of document *A2* to pull data from, say, an updated *A1₄* into their schema. This might cause a species for which they had previously selected an image to fall out of the rankings, and be replaced for another, a piece of data loss they would have to patch up manually.

4.2. “Impute Version” action

A variety of the “Make Relation” action, that we call “Impute Version”, can occur when a document is imported into the system from the outside, e.g. when received as an emailed spreadsheet, and the document is to be considered an updated or otherwise variant version of one already in the system. Normally, all the “lenses” involved in this imputation will be the null lens, but if the data involved is not

well supplied with unique keys or there has been some other kind of disruption to its structure, the work of mapping the rows of the updated data onto the original may not be trivial. Just as with the ordinary “Make Relation”, we expect the system to apply brilliant AI, or else draw on a repertoire of canned responses previously found useful in the community, to ease importing the data back into alignment. An example of such an action would be the arrival of document AB_2 in the section 2 system — this is reasonably well-aligned with earlier versions but in practice required significant manual effort to check and align.

4.3. “Break Relation”

The correlate of the “make relation” action is required when a community’s direction seems to take it away from alignment with one of its previous neighbours. Our small-scale example doesn’t span enough time or space to encounter this, but one could imagine this dynamic emerging, for example, with the taxon resolution databases in section 3. Two communities may have been content to share a database for a significant time, until at some point either a change of mission, or the discovery of data anomaly imposes on them the burden to start maintaining separate databases. At this point it should be easy both to simply “fork” a pre-existing resource into two which may then continue to diverge, but also to retroactively “unpull” any data which had previously been pulled over a previously existing relation to restore it to the value it would have otherwise had, without disturbing any other data in the system.

4.4. System Infrastructure

We’ve spoken loosely of “the system” in previous sections but, following the initial discussion in this section, it’s clear that it could not be incarnated as either a purely localised or a purely distributed system. Rather than being a “walled garden” system as is operated in typical data notebook systems, data is not only imported into the system from the net at large, but is also automatically exported back into it at easily discoverable URLs. For every document at every point in the system, the interface allows easy to a public URL from which an ordinary CSV for that document’s state can be read. However, alongside this export must similarly be an encoding of the metadata not only for any schemas in the data, but also of the system’s tracking of relations to related data. This will be done “by convention” — given a particular URL scheme, the URL for the metadata can be statically deduced from the URL for the CSV. If the metadata is not found at that URL, then the data will be imported by copying into the user’s “local node” of the system, with an annotation in its metadata encoding where the data was found in the wild.

5. Conclusion

We’ve walked through a number of real-world collaboration problems, and exhibited numerous use cases that are poorly met by existing office-type tools, and all others that are currently available to our target communities at a price they can afford. Our design sketch for a family of solutions needs considerable refinement down to detailed user interactions and detailed implementation specifications, but we believe represents the blueprint of a relatively tractable implementation effort, and one that could ease the huge burden of articulation work still felt by cooperating communities sharing related but discrepant data. We’ve stressed the importance of operating a rarely seen idiom, which we’ve named “forgiving relations”, under which the system indefinitely tolerates data which does not satisfy relationships which have been encoded into the system.

6. References

- Basman, A., & Tchernavskij, P. (2018). What Lies in the Path of the Revolution. In *Proceedings of the Psychology of Programming Interest Group*.
- Foster, J. N., & Pierce, B. C. (2009). *Boomerang Programmer’s Manual*. Retrieved from <http://www.seas.upenn.edu/~harmony/manual.pdf>
- Stace, C., & Thompson, H. (2019). *New Flora of the British Isles* (4th ed.).
- Tchernavskij, P., Basman, A., Nouwens, M., & Beaudouin-Lafon, M. (2019). Control and Ownership of Artifact Ecologies in a Biodiversity Research Network. In *To appear: Proceedings of the 22nd ACM Conference on Computer-Supported Cooperative Work*.

Evaluating programming systems design

Jonathan Edwards
 jonathanmedwards
 @gmail.com

Stephen Kell
 University of Kent
 S.R.Kell@kent.ac.uk

Tomas Petricek
 University of Kent
 T.Petricek@kent.ac.uk

Luke Church
 Computer Laboratory
 University of Cambridge
 luke@church.name

Abstract

Research on programming systems design needs to consider a wide range of aspects in their full complexity. This includes user interaction, implementation, interoperability but also the sustainability of its ecosystem and wider societal impact. Established methods of evaluation, such as formal proofs or user studies, impose a reductionist view that makes it difficult to see programming systems in their full complexity and, consequently, force researchers to adopt simplistic perspectives.

This paper asks whether we can create more amenable methods of evaluation derived from existing informal practices such as multimedia essays, demos, and interactive tutorials. These popular forms incorporate recorded or scaffolded interaction, often embedded in a text that guides the reader. Can we augment such forms with structure and guidelines to obtain methods of evaluation suitable for peer review? We do not answer this question, but merely seek to identify some of the problems and instigate a community discussion. In that spirit we propose to hold a panel session at the conference.

1. Introduction

The ability to disseminate knowledge, which relied on the invention of the printing press, was essential for the birth of modern science in the 17th century¹. The internet changed how we distribute academic papers, but their format has changed surprisingly little. Despite new media formats that the internet enables, our research is still largely presented in the same format as in the 17th century.

There are some noteworthy exceptions. An increasing number of academic publications that rely on computational methods publish "software artifacts" in addition to a paper which, at least in principle, allow others to repeat the experiments. Grossman et. al. (2016) propose embedding "animated figures" into research papers through multimedia features of PDF documents. Some authors create *interactive essays* or *explorable explanations* (Brusilovsky 1994, Victor 2011), which are articles that the readers can interact with in order to better understand the presented subject, for example by modifying input parameters of a computation and seeing how this affects the results. Those cover topics ranging from social science, e.g. (Hart and Case, 2014) to machine learning, e.g. (Carter et al., 2016), but they are typically educationally focused and do not present new research results.

We argue that we need to establish new media for presenting novel research ideas on programming systems design. We want to make such new media a core method for presenting some aspects of our work. Finally, to put such new media on a par with other ways of presenting ideas on programming, we want to find rigorous evaluation methods for such new media presentations.

Research on programmer experience needs to explore the capabilities that a system design offers to a programmer, how those affect programmers conception of the problem at hand and how it changes the way they conceptualize the world they need to model and interface with. Traditional static paper format is unsuitable for exploring such questions. A new media format we envision makes the reader

¹ As documented by Wootton (2015), the printing press made knowledge public and allowed a wide-spread exchange of ideas among, for example, astronomers belonging to different cultures.

more naturally ask the aforementioned questions and it is also more suited for discussing systems that go beyond the textual program representation.

The aim of this paper is not to present a final new media alternative to the static paper format, but rather, to start and stimulate a discussion on this topic. In order to do this, we present a range of different positions that are rooted in different backgrounds, have different motivations and argue for different methods. Although the focus of individual positions differs, we structure each statement in a similar way. We hope that this format adds a level of coherency between individual positions, but more importantly, will also encourage others to contribute to the debate that we are hoping to start with this paper and with a proposed panel session at the conference.

2. Multimedia essays: Evaluating experience design

One important kind of research is about *designing* programming systems to improve their human *experience*. We italicize the words *designing experience* because they are a problem: they make it difficult to report research contributions using traditional academic papers, for two reasons. First, papers restrict explanation of interactive experiences to static figures and screenshots. Secondly, the norms of academic papers are not well suited to discussing design issues, which fit neither into reporting empirical results nor proving theorems.

Is there a better way to report novel designs of system experiences? We propose to adapt an established informal practice: multimedia web essays (primarily text containing video clips) for this purpose. The challenge is to make this form rigorous enough to be used in a peer review process. Our approach to this challenge is to impose structure and guidelines, for both the authors and reviewers.

2.1 Why does it matter: Evaluating live programming research

Consider the nascent field of Live Programming (Tanimoto 1990, LIVE 2013). This field aspires to make programming more like “spreadsheets” by synthesizing a visual substrate offering immediate concrete feedback at all times. Such research is a matter of co-design of a PL and IDE, rethinking the nature of both so that in combination they offer a live programming experience. This is not PL research, as it is not about formally verifiable properties. Neither is this HCI research, as it is not foremost about observable user behaviors, but rather about the conviviality of conceptual models and visualizations.² As a result it is difficult to publish research on Live Programming in either PL or HCI venues. More insidiously, publishing in such venues can lead us off course by forcing us to answer the wrong questions.

On the other hand, venues like the LIVE Programming Workshop (LIVE 2013) accept work presented informally, through screencasts and verbal presentations. But lacking written exposition that can be carefully reviewed, it can be hard to ascertain what contributions these submissions make. Also lacking discussions of related work, it can be hard to situate such contributions as a conversation among researchers, which we feel is crucial to making progress as a field.

We need to find some way to report our research that is suitable for discussion of system experience design while also being rigorous enough to support the proven norms of research: peer review, archival publication, and citation.

2.2 What are the problems: Evaluating design

Our goal is to improve the human experience of using and developing software. This goal is inherently: unformalizable and unquantifiable. Therefore the discussions we need to have are about *design*: discovering technical choices offering a propitious balance between conflicting and sometimes subjective qualities.

² We agree in principle that a mature Live Programming system should be validated by randomized controlled trials, but that is wholly impractical at this stage of the field, and we see little evidence that this method has succeeded even in more mature fields of software research.

We need a way to evaluate the design of systems from the viewpoint of their human experience, and we need this to be lightweight enough to discuss early-stage work. Video is the natural medium for demonstrating user experiences. But carefully crafted prose is still the best medium for explaining ideas in detail to be studied and reviewed by others. These requirements match an emerging form of discourse on the web: multimedia essays consisting of text with video clips. An influential early example of this form is Bret Victor's Learnable Programming (Victor 2012), but see also (Petricek 2016). These essays tend to be informal and introductory, addressed to an audience of interested observers rather than peer researchers within the field. To serve as a medium for reporting and evaluating research contributions, more rigor is required.

2.3 How to address those problems: Tentative guidelines

Every field of research adopts certain methods for reporting and evaluating contributions, and in many ways these methods define the boundaries of the field. These methods strike a contract (often unwritten) between authors and reviewers. We propose to adapt multimedia essays into a method of evaluation by making this contract explicit as a set of guidelines for both authors and reviewers, backed by normative examples.

Many forms of evaluation become rigorous by setting high standards for authors to substantiate their contributions. Instead we shift part of the burden onto the expert reviewers, who are asked to use their best judgement to answer the question: would it benefit the research community to publish this idea?

Guidelines for authors: What makes for a rigorous multimedia essay

1. The essay identifies a targeted domain of software, and a targeted profile of user/developer.
2. The essay describes the state of the art and enumerates specific contributions that are claimed to be novel solutions with benefits over known techniques. Related work is cited either in context or a separate section.
3. Each contribution is categorized as either a technical novelty in some field (eg PL) or as a novel design combining or unifying known techniques (within or across fields). Each contribution states what benefits it is claiming, categorized by taxonomies such as Cognitive Dimensions (Green&Petre 1996) or Olsen (2007, summarized by Hempel 2019).
4. Each contribution is discussed in terms of one or more authentic examples from the problem domain. The benefits over known techniques are substantiated by comparison with their application to corresponding examples. Benefits to user experience are demonstrated with video clips, where possible also for the comparable known technique.
5. Each contribution discusses its implementation: is the demonstrated experience a mockup or an implementation? What is novel about the implementation? What is the hard part? How general is it—how would it handle variations on the demonstrated scenarios?
6. Design deficiencies and unsolved problems are disclosed, preferably in terms of problematic examples, and especially problematic variations upon the examples used to claim benefits. As a rule of thumb, there should be about as many problematic examples as beneficial ones.
7. Discussion of mistakes made, lessons learned, and suggestions for future research are all encouraged.

Guidelines for reviewers: How to recognize important contributions

1. Unless the submission claims to be a survey, reviewers should only expect familiarity with significant related research, not encyclopedic knowledge.
2. New ideas can seem obvious in retrospect. Assuming that the essay shows general familiarity with the field, if a reviewer believes a contribution is trivial or already known then it is their duty to substantiate that position (see Olsen 2007).
3. Offer constructive criticism, about both form and content. In addition to technical issues, it can be very helpful to note unclear prose and the need for examples. Distinguish between two kinds of criticism: criticism of the evaluation methods used vs. criticism of the ideas themselves. Weight these two dimensions appropriately for the venue.

4. Answer this question: in your expert judgement, would it benefit the research community to publish this submission? In other words, is it worth reading?

2.4 What will this allow: Towards humane technology design

A better medium in which to report and evaluate our research will speed and tune the intellectual feedback loop that leads to progress, and sometimes even breakthroughs. In the longer term, establishing a distinct methodology can help us mature into a fully formed field. More broadly, we see ourselves as part of a wider movement toward *humane technology design*, and we hope that these methods might help others join together into productive research communities.

Next steps: Building examples and growing community

1. Solicit feedback from the community—indeed it is the purpose of this paper to initiate that conversation at the Psychology of Programming Interest Group (PPIG) meeting.
2. Do trial runs with our own work, leading to a more refined and detailed proposal.
3. Hold a workshop inviting submissions using our proposed methods, as well as meta-discussion of the methods.

3 Interactive essays: Asking new questions about programming

In this section, we present an argument in favor of *interactive essays*. This position shares many aspects with the previous one. However, it additionally requires interactivity that allows the reader to use some aspects of the presented system in some, possibly limited, ways. The interaction is guided by the essay author - the mixed content format allows the author to first introduce design principles and then lets the reader get a glimpse of what the envisioned experience looks like.

The most important aspects of this proposal are that, although the narrative is controlled by the author, the reader can experience some aspects of the system live and that the reader can also deviate from the narrative suggested by the author and explore the boundaries of the actual prototype or demonstration implemented by the author.

An example of such interactive essay is an introduction to the theory of coeffects by Petricek (2016). This example is more educational and it is centered around programming language theory research, rather than programming experience, but it follows the format that we argue for here: it allows the reader to interact with a (somewhat limited) system within a narrative designed by the author.

3.1 Why does it matter: New media for new thoughts

Present ways of publishing and evaluating programming research are not doing a good job:

- A paper presenting programming systems design research can easily be rejected for the lack of evaluation. This is not because the authors did not want to evaluate their work, but because we do not have established standards for evaluating such work.
- A paper focusing on interaction or user experience is often treated as "early work". This is because we assume that the idea is not yet mature enough to be, say, formalized and presented in the form of mathematical properties.
- Consequently, authors often change the focus of their work and start studying formal or empirical aspects that are easier to evaluate. In doing so, they abandon the aspects that made the work interesting in the first place.

A new media format that allows us to effectively present user experience or interaction with a programming system has the potential to change this. If we succeed at finding a way of rigorously evaluating such contributions, we will be able to focus on studying how programmers interact with systems and how new ways of interaction can better support programmers in their job. We will also be able to better understand and extend other work in this space, building a flourishing community of programmer experience researchers.

3.2 What are the problems: Rigorous communication and evaluation tool

To make interactive essays an effective research communication and evaluation tool, we need to change the perception surrounding interactive essays, find rigorous ways of reducing complexity of systems and tackle development and archival issues.

From education to research

Recent work on interactive essays has been focused on education teaching, for example, how a certain model or algorithm work. This is useful, but it does not address our needs. If we are to use new media to publish novel research ideas, we need to find a suitable method of structuring the ideas. An example of such structuring from formal methods is the idea of *structured operational semantics* (Plotkin and Kahn, 1987), which allowed a growing number of people to present new programming language constructs. Structured operational semantics defines a common way of reasoning about the correctness of a programming language, but it can be applied to topics ranging from concurrency to network communication. Similarly, we need a structure of interactive essays that defines a common method, but can be used for a wide range of programming experiences.

Finding new way of reducing complexity

When evaluating programming systems, we might hope to implement the whole system, use it for a non-trivial task in a real-world setting and reflect on the experience and wider societal implications of the system. This is infeasible and so we need to reduce the complexity of the task in some way.

Different evaluation methods ignore different aspects of such complexity. For example, formal proofs only look at small models ignoring all practical aspects. How can we simplify the presentation of a user experience so that we still capture important aspects of the desired user experience and, ideally, convey this to the reader or viewer in a direct way? Do we need to make our presentation interactive enough that a reader can feel what using the system would actually feel like?

Rigorous evaluation of user experience

In order to be able to evaluate such presentation of programming ideas, we also need to understand what counts as a rigorous presentation of a programming system in an interactive new media format. We have a good understanding for theorems involving mathematical properties and for empirical evaluation, but what are the standards of rigor for interactive illustrations of user experiences?

Development and archival issues

The final challenge of presenting user experiences in an interactive format is that, compared to a traditional paper, such presentations are significantly harder to develop and archive. There is a range of options. A static article with embedded screencasts (as discussed in Section 2) is relatively easy to produce, but it might be too guided to effectively convey the programmer experience. A partly functioning system that runs in a web browser is more complex and may convey the idea more clearly, but it is only usable for certain kinds of programming experience research. Finally, web-based essays will inevitably be more difficult to archive. This is easier to address if they are self-contained, but can get increasingly difficult if such essay uses external services or data sources.

3.3 How to address those problems: Methodologies, community and technologies

Making interactive essays an established way of presenting and evaluating programming system design poses technical, academic and social challenges. For this reason, addressing the problems outlined above requires new technologies, new academic standards and also a new community.

Extensive methodology section

It will take time until the community producing interactive essays settles upon standards of rigor. This will most likely take the form of tacit unwritten knowledge, as identified by Polanyi (2012). Until such standards are widely understood, we need to include an explicit and detailed methodology section in our publications. This needs to clarify what method the essay follows and which aspects of it should be evaluated in what ways. For example, when a new system design is presented, one can claim that it is interesting for its novelty. This can be supported by a comprehensive comparison with

related work. Alternatively, one might claim that a certain theory allows an interesting range of programmer experiences. This can be supported by a prototype implementation of some of those and an outline of work that needs to be done for other cases. By having an explicit methodology section, the readers and reviewers will know how the key contributions of the essay are supported.

What makes new media rigorous

An interactive essay should share many common properties with standard academic papers. It should have a methodology section that is often missing in computer science papers, discussion of related work and clear statement of contributions. An interesting problem is how to judge the rigour of the interactive aspects of the essay. We offer several suggestions:

- The essay should outline how can the proposed system design be implemented. Essays based on a prototype implementation should document what has been omitted and how it could be completed. Essays not including implementation need to argue in another way.
- The essay should illustrate some interesting “effect” such as a new, appealing user experience story. This can be documented by an interactive walk-through. A rigorous essay should also document how this experience arises from a more basic design principles and how those principles extend to other aspects of the system.
- In order to develop a flourishing community, system design essays need to have common points on which they can be contrasted. Those can be common programming challenges or domains - a rigorous essay will aim to include user experiences that make it possible to relate the presented ideas to existing work.

3.4 What will this allow: Changing the questions we ask

The way we evaluate our research has an effect on what research we conduct. According to present day standards, programming systems design research often appears as early work. It rarely comes with proofs, detailed empirical evaluations or large scale user studies. However, adding any of these often shifts the focus of the work away from the original focus on *programming systems design*.

If we can (i) make interactive essays a standard format for presenting programming system design research, (ii) build a community around such format and (iii) agree on standard of rigour for such essays, then we will be able to move the field of programming systems design forward. The format will make it possible to evaluate the design of such systems from a more comprehensive perspective. It will allow producing research publications that share enough of their problems and methods so that they can be compared and inspire others to contribute new ideas to the common ecosystem.

4 Papers plus: Letting the format suit the work

If we want to transcend the limitations of paper, we should be careful not to transcend its benefits too. In this section, we outline some of the benefits of traditional paper formats and discuss how new media formats can keep the good properties of well-written static papers.

4.1 Why it matters: Evaluable and durable formats for evaluation by inquiry

A clear motivation for exploring such alternatives is in the spirit of ‘the medium is the message’ (McLuhan, 1967) and the potentially helpful disruption that new media may bring. For example, inclusion of new media may usefully break ice between researchers and (other) practitioners. Lacking a formalised or “scholarly” culture, communicative practitioners exploring new ideas have already pursued a broad range of techniques, from web-based written documents that in some cases resemble papers, through video essays, interactive demos all the way to simply releasing software.

The primary advantages of most ‘new media’ approaches has been some mixture of explanatory effectiveness and support for ‘evaluation by inquiry’—active investigation on the part of the reader or consumer, rather than simply a report by the authors. There is clear value to these, but also the clear challenge: how to make them sufficiently evaluable and durable that they might take a place alongside traditional papers.

4.2 What are the problems: holistic and versatile reporting

While gaining in explanatory effectiveness, we would also like to retain some abilities: to talk holistically and comparatively, and to deal in relatively evaluable artifacts. There is also a latent question of how prescriptive any submission format should be: work is diverse, so progress relies on enabling a sufficiently diverse range of ‘shapes of submission’. Certain new risks also emerge: for example, papers can be anonymised and otherwise curbed in respect of the prejudices they might elicit, whereas, for example, audio and video present inherent problems if they are fronted by a human face or voice.

Versatility of academic papers

Papers are versatile: they offer a viable way to walk through arguments, methods, designs, experiences and other findings. They are inherently fairly accessible to readers (requiring little fancy software or equipment to experience), fairly durable, and extend to a huge range of topics, not only in subject matter but in modes of contribution (position papers, empirical studies, design rationales, surveys, and so on).

Conveying design holistically

One specific challenge in not-just-paper modes, especially anything resembling a demo, is to convey a design holistically. Programming systems research often has relatively abstract goals, such as to improve the comprehensibility or factoring of a certain kind of code, to improve the economics of a long tail of currently tedious programming tasks (e.g. integration), or to introduce some elaboration of an area that is currently poorly understood or ill-characterised. In respect of these, demo-like vehicles may struggle: since a demo focuses at all times on presenting something specific, it risks an over-narrow presentation which conveys one de-emphasises breadth. *Showing* a long list of examples, while possible, is harder than merely naming or describing them. Showing a smaller number and simply naming others (e.g., in narration) is perhaps viable in some cases. In others, a paper may still be the best option. Holism is difficult to account for: human beings’ ability to generalise and to imagine is sometimes most effectively triggered by a well-conveyed example, but equally, details can distract from the big picture.

Retaining comparison

Since a demo has (one assumes) a unilateral focus on ‘the’ artifact being discussed, it makes more difficult the comparative mode of exposition, on which most research papers rely heavily to convey both novelty and essence. The de-emphasis of comparators is already a risk in papers: a common pattern in poorly written papers about practical work is to focus on ‘the thing we built’ without addressing the question of how it differs from existing systems and hence why it should be accepted as novel and valuable research.

Realising one’s comparators in a demoable form, while possible, is harder than naming or describing them. To make a comparative point, focus on ‘a demo’ of ‘the artifact’ must either be paused, or must embody the alternative—a juxtaposed ‘demo within a demo’!—to handle comparative questions. Any expectation to do this may raise the effort bar unreasonably high. On the other hand, making this effort has obvious value in encouraging like-for-like comparison and reducing the risk of mischaracterising the prior work. In effect it amounts to a reproduction study.

Effort, benefits and costs of new media presentations

Another concern is balancing effort, benefit and credit. While achieving the above-mentioned ‘difficult but possible’ feats in demo-like form is certainly desirable in principle, the effort and reward may or may not be proportionate. Depending on the other contributions on offer, they may or may not fit into a reasonably sized and shaped ‘unit of contribution’. One of the strengths of papers is that they allow some flexibility in these sizes, shapes and proportions. The goal should be to admit more flexibility, not less, while retaining a cost-effective process from submission to review to publishing. Just as ‘possible, but difficult’ suggest greater demands on the author, ‘evaluation by inquiry’

arguably makes more demands of the reviewer, relative simply to reading a paper and digesting its argument (already a considerable task!).

4.3 How to address those problems: Presenting a structured argument

A possible guiding principle might be to oblige authors to present a structured argument as the primary object of evaluation, as with papers. Inquiry-based evaluation may be offered as an additional mode by which the evaluator may satisfy themselves of that argument. Authors who prefer to ‘let the system do the talking’, may do so to an extent, without pushing great responsibility for initiative onto the evaluator. A common ingredient in all forms of submission should perhaps be a linearised presentation of this argument---in short, it should be possible to ‘press play’, even if the artifact permits inquiry or interrogation. The more interrogatable an artifact, the more contribution is potentially apparent from that, as opposed to from writing.

In light of the issues of prejudice and anonymisation identified earlier, a policy of ‘no voices, no visages’ may be preferable (e.g., subtitles rather than voiceover).

In some communities, artifacts or tools by themselves are an object of independent recognition. Some conventions are therefore required on what is covered by a given submission versus what aspects may be redeemable later for ‘extra credit’ by a submission with different emphasis. The right conventions are unlikely to be easily anticipated; observing the emergent patterns is necessary.

4.3 What will this allow: Tailoring formats to the needs of the work

If done right, an approach will allow authors to tailor a format to the needs of the work, mixing some amount of new media (video, audio, web-like interactive features, or something yet unanticipated) while retaining a clear argument. If such a format can succeed, it may break the ice, without breaking the glass: allowing a wide range of work to gain credit and recognition in a publication-based system, from diverse participants and backgrounds, while still allowing fair and thorough evaluation. Accommodating a continuum, with papers remaining a point thereon, may prevent dismissive attitudes or ‘ghettoisation’ of work choosing these new approaches.

5 Conclusions

When writing a programming system paper, we have various ways of evaluating our work. We can study its mathematical properties and publish proofs, we can measure our implementation and publish charts or we can run user studies and publish quantitative or qualitative summaries. A static academic paper is a great format for presenting such evaluation, possibly with a software artifact for empirical measurements. But are our preferred evaluation methods determining our publication format and research focus, or is our publication format and research focus determining our evaluation methods?

In this paper, we argued that we need to establish new media for presenting novel research ideas on programming systems design. Our aim has been to stimulate a discussion. To this end, we presented three different positions that are inspired by different research problems and argue for somewhat different methods. We discussed a range of challenges that such new media presentations are facing, possible ways of addressing those and our thoughts on how the world of programming systems research would benefit from adopting such new formats.

5.1 A range of technological solutions

We presented three concrete positions in this paper, arguing for media ranging from traditional papers, suitably augmented with other media to interactive web-based essay that run a prototype of the envisioned system directly in a web browser. One tentative conclusion of this paper is that we should accept and welcome a range of presentations including:

- An *interactive essay* combines text that provides a narrative in the similar sense in which a traditional academic paper does with interactive components that show aspects of the discussed system in action. This is an ambitious goal as it involves not only producing a prototype implementation, but also orchestrating the interaction with the reader in the essay.

- In a *screencast essay* with embedded videos, each screencast documents one particular interaction. It does not let the reader choose a different path, but it does let the reader see what the actual interaction with the programming system looks like. We should accept screencasts that are based on an actual implementation or mock-ups, but we need to acknowledge the difference as part of the methodology section included in each publication.
- An essay based on videos can be made more interactive by offering the reader a choice of several predefined alternatives as in *choose your own adventure* games. This is similar to wireframing tools known from user interface design, but we should not follow wireframing tools in the level of abstraction they use. Our goal is to provide full record of user experience.
- *Scrollytelling* (Seyser and Zeiller, 2018) refers to online articles that present a story, which unfolds as the reader scrolls through the article. Scrollytelling articles often combine text, where the reader just scrolls down, and interactive sections where scrolling takes a different role and allows the reader to move back and forth through an video-like visual element³. In its basic form, scrollytelling allows the viewer to easily go backwards and replay parts of “story” that they are interested in. It could also be combined with an interactive essay, which would let readers scroll to follow the author’s narrative, while allowing them to take a different course of action to explore other aspects of the system.
- *Onboarding* refers to a range of UI design patterns for training new users with minimum friction (UI-patterns 2019). These patterns scaffold or restrict the UI of an application in order to guide the user’s initial experience along a happy path. These same methods could serve as a guided tour of a research prototype intended to showcase to reviewers novel contributions and design benefits, while bypassing unimplemented regions.

5.2 Philosophical reflections and scientific experimentalism

The fact that we now largely view programming as manipulation of linguistic entities is not because that would be the only way of programming, but rather, a consequence of historical developments that made programming language a *language* in the 1950s (Nofre et al., 2014). We believe that programming systems design research that we advocate in this paper needs to take a different approach. However, the ubiquity of the linguistic metaphor means that much of our academic infrastructure, including evaluation methods, is designed around it. The transformation of our research focus and of our research techniques can only proceed hand in hand. Interactive essays, as discussed in this section, are one of the aspects that can support this paradigm shift.

In the current paradigm, the central point of our knowledge is ‘how’. Theoretical papers capture the formal essence of ‘how’, while empirical papers discuss practical implementation aspects of ‘how’. If we adopt interactive essays as our way of disseminating results, we will be able to also build and grow knowledge focusing on ‘what’. That is, what are the different user experience that a system based on a particular design can provide?

In philosophy of science, this attitude is akin to the experimentalist approach of Ian Hacking (1983). Hacking discusses problems with many accounts of scientific progress and proposes an alternative - science progresses by accumulating practical experimental knowledge. Developing a theory of electrons and positrons is not (yet) such knowledge, because our interpretation of what an electron or positron is might (and frequently does) change. However, learning how to change the charge of a niobium ball by spraying it with positrons or electrons a different kind of knowledge. Even if our theories change so that it does not even involve electrons and positrons as primitive entities, we can still conduct the practical experiment and new theories will have to find new explanations for it.

Similarly, a significant amount of programming systems knowledge is heavily reliant on the theory within which it was developed (an efficient way of compiling purely functional languages is not interesting to anyone using another kind of programming language). This should not be the case with

³ For example, see the New York Times article by Almukhtar and Watkins (2016), where scrolling controls a timeline of an event illustrated using an animation on a map.

experiences documented in the form of interactive essay. Even if we start thinking about programming systems differently or use a different implementation method, we can still follow the documented user experience. We will either find a way of replicating it in a new system, or we will know that we are losing something valuable.

Acknowledgements

We thank for helpful comments: Jun Kato.

References

- Almukhtar, S. and Watkins, D. (2016). How One of the Deadliest Hajj Accidents Unfolded. New York Times, Available online (retrieved 8 June 2019):
<https://www.nytimes.com/interactive/2016/09/06/world/middleeast/2015-hajj-stampede.html>
- Brusilovsky, P. (1994). Explanatory visualization in an educational programming environment: connecting examples with general knowledge. In International Conference on Human-Computer Interaction (pp. 202-212). Springer, Berlin, Heidelberg.
- Carter, et al. (2016). Experiments in Handwriting with a Neural Network, Distill
<http://doi.org/10.23915/distill.00004>
- Green, T. R. G.; Petre, M. (1996). Usability analysis of visual programming environments: A 'cognitive dimensions' framework. Journal of Visual Languages and Computing. 7: 131–174.
- Hacking, I. (1983). Representing and intervening: Introductory topics in the philosophy of natural science. Cambridge University Press
- Grossman, T., Fanny Chevalier, and Rubaiat Habib Kazi (2016). Bringing research articles to life with animated figures. ACM Interactions 23, 4 (June 2016), 52-57.
<http://interactions.acm.org/archive/view/july-august-2016/bringing-research-articles-to-life-with-animated-figures>
- Hart, V.; Case, N. (2014) Parable of the Polygons: A Playable Post on the Shape of Society Available online at: <https://ncase.me/polygons/> (retrieved 7 June 2019)
- Hempel, B. (2019). Summary of Olsen's Evaluating User Interface Systems Research.
<https://people.cs.uchicago.edu/~brianhempel/Evaluating%20User%20Interface%20Systems%20Research%20-%20Graphical%20Summary.pdf>
- LIVE (2013) Proceedings of the 1st International Workshop on Live Programming. ICSE 2013
- McLuhan, M.(1967). Medium is the massage. An inventory of effects. Penguin Books.
- Nofre, D., Priestley, M., & Alberts, G. (2014). When technology became language: The origins of the linguistic conception of computer programming, 1950–1960. Technology and Culture, 55(1), 40-75.
- Olsen, D. R. Jr. (2007). Evaluating User Interface Systems Research. UIST 2007.
- Petricek, T (2016) Coeffects: Context-aware programming languages. <http://tomasp.net/coeffects/>
- Plotkin, G. and Kahn, G. (1987). A structural approach to operational semantics. In proceedings of STACS'87, pp. 22-39. Aarhus University.
- Polanyi, M. (2012). Personal knowledge. Routledge.
- Seyser, D. and Zeiller, M. (2018). Scrollytelling—An Analysis of Visual Storytelling in Online Journalism. In 22nd International Conference Information Visualisation (IV). IEEE
- Tanimoto, S. (1990) VIVA: A visual language for image processing. Journal of Visual Languages and Computing, Vol. 1, No. 2, June 1990.

- UI-patterns (2019) Onboarding <http://ui-patterns.com/patterns/onboarding/list> retrieved June 2019
- Victor, B. (2012). Learnable Programming. <http://worrydream.com/LearnableProgramming/>
- Victor, B. (2011). Explorable Explanations. Available online at:
<http://worrydream.com/ExplorableExplanations/> (retrieved 7 June 2019)
- Wootton, D. (2015). The Invention of Science: A New History of the Scientific Revolution. Allen Lane Books. ISBN 978-1846142109

Open Piping: a Visual Workflow Environment

Charles Boisvert
 Sheffield Hallam University
 c.boisvert@shu.ac.uk

1. Introduction

This submission presents our work on *Open Piping*¹, a visual workflow environment to make functional programming and data handling accessible to inexperienced learners.

2. Motivations

Four elements motivate our work: the growing ease of use and learning of programming tools; the rise of big data and data analytics, underpinned by functional programming; the visual model of functional computation; and finally the access barriers to this programming paradigm and to data science.

2.1. A systematic improvement in access to programming

Usability breakthroughs mark the progress of all computer science, including programming. One remarkable advance is the wide range of programming learning and novice developer environments, using a jigsaw puzzle metaphor to represent individual statements, such as MIT Scratch (Resnick et al., 2009).

2.2. The rise of data processing

While simple applications have become more accessible, computation has shifted to new domains, and to programming languages that support multiple paradigms, like R, Clojure, or Python which add functional programming to imperative, object-oriented and event based development. Yet, the jigsaw puzzle metaphor favours an imperative perspective on programming: the programming paradigms computing education tools support best, are becoming less used in professional practice.

2.3. Modelling functional computation visually

Lambda calculus' mapping to directed acyclic graphs provides a visual model, summarised table 1. The graph, or boxes-and-wires (hereafter BW) model, can read as a data flow.

Notation	Represents	Graphical equivalent
x	Variable	
$\lambda x.f$	Abstraction (function f has parameter x)	
fx	Application (function f is applied to variable x)	

Table 1 – Basic elements of untyped λ -calculus and their representation as boxes and wires (BW)

2.4. Access limitations to visual functional programming

Data analysis applications require mastery of complex systems to apply mathematical techniques and represent information in non-trivial domains. Users', particularly novices, need carefully designed presentation and interaction devices. Below, we consider some variations that have been attempted.

3. Visual design

Coordinating code with result. The BW model represents computer code, but many visualisations offer multiple coordinated views of results, as Shneiderman and North (2000) propose. Yahoo pipes²

¹<http://boisvert.me.uk/openpiping>

²Discontinued by Yahoo, Inc. Wayback machine archive: <https://web.archive.org/web/20150604234337/http://pipes.yqlblog.net>

co-ordinates code with a sample of the resulting data. Selecting subsets of code supported debugging.

Data Typing. Viskell shows textual type annotations, and PROGRAPH visual shape clues, as seen in fig. 1 below. Yahoo pipes did not show types, but enforced type checking through interaction.

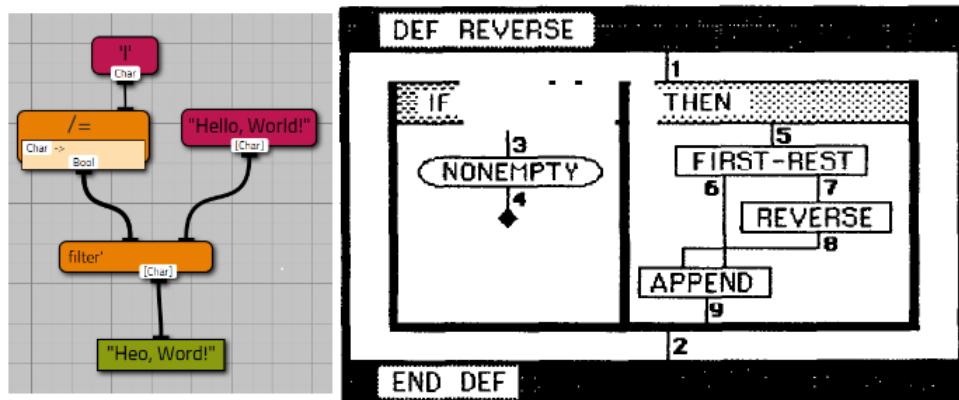


Figure 1 – BW in Viskell, left (Wibbelink, 2016) and PROGRAPH, right (Cox and Mulligan, 1985)

Representing Conditionals. A conditional execution function directly maps to a box and three wires, but this 'direct' solution may not be 'directly' understood by end-users. PROGRAPH uses a *frame* to divide code in chunks that end-users best consider separately.

First class functions. First-class functions' representation in the model is clear: a box represents a function, unless a wire shows its application to a variable. But as Viskell's use of this solution (pictured above) shows, a novice would not understand it without help. An alternative is to represent first-class functions within frames. Fukunaga *et al.*'s (1993) discusses a full representation of first-class functions with this solution. It shows the need for careful study to represent the notions in ways that users can understand and control.

4. Conclusion

We believe that Data Processing is inaccessible to the public, mainly due to a cognitive barrier, but that existing work shows ways to effect greater ease of learning, of use and availability to the functional languages and data analysis tools. To make the visual model support development by end-users, learning by novices, understanding by learners, we will need to propose and evaluate alternative solutions empirically.

References

- Cox, P. and Mulligan, I. (1985). Compiling the graphical functional language program. In *Proceedings of the 1985 ACM SIGSMLL symposium on Small systems*, pages 34–41. ACM.
- Fukunaga, A., Pree, W., and Kimura, T. D. (1993). Functions as objects in a data flow based visual language. In *Proceedings of the 1993 ACM Conference on Computer Science*, CSC '93, pages 215–220, New York, NY, USA. ACM.
- North, C. and Schneiderman, B. (2000). Snap-together visualization: A user interface for coordinating visualizations via relational schemata. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '00, pages 128–135, New York, NY, USA. ACM.
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al. (2009). Scratch: programming for all. *Communications of the ACM*, 52(11):60–67.
- Wibbelink, F. (2016). Interacting with conditionals in viskell.

Constructing a Model of Expert Parallel Programmers' Mental Representations Formed During Parallel Program Comprehension

Leah Bidlake

Faculty of Computer Science, Faculty of Computer Science, Department of Psychology
University of New Brunswick
leah.bidlake@unb.ca, aubanel@unb.ca, voyer@unb.ca

Eric Aubanel

Daniel Voyer

Abstract

Parallel programmers are frequently tasked with modifying, enhancing, and extending parallel applications. To perform these tasks and maintain correctness, parallel programmers must understand existing code by forming mental representations. The comprehension of parallel code requires programmers to mentally execute multiple timelines that are occurring in parallel at the machine level. The goal of the proposed research is to develop a model for parallel program comprehension. The study will investigate the mental representations formed by expert parallel programmers during the comprehension of parallel programs. The task used to stimulate the comprehension process will be verifying the correctness of parallel programs by determining the presence of data races. Eye tracking data and questionnaires will be used to formulate a model.

1. Introduction

During the comprehension process, programmers form mental representations of the code they are working with (Détienne, 2001). Understanding these representations is important for developing programming languages and tools that enhance and assist programmers in the comprehension process and other tasks. The cognitive component of program comprehension that is of interest here is the abstract mental representations that are formed during program comprehension. These mental representations, often referred to as mental models, are founded in the theories of text comprehension (Pennington, 1987). The mental model approach to program comprehension is based on the propositional or text-based model and the situation model that were first developed to describe text comprehension (Détienne, 2001).

2. Research Goals

In parallel programming there is a significant lack of theory to inform the development of programming languages, instructional practices, and tools (Mattson & Wrinn, 2008). Empirical research on mental representations formed by programmers during program comprehension has been predominately conducted using sequential code. Studies involving parallel programmers are most often concerned with productivity (Hochstein et al., 2005; Ebcioğlu et al., 2006).

The comprehension of parallel code requires programmers to mentally execute multiple timelines that are occurring in parallel at the machine level. Therefore, parallel program comprehension may require additional dimensions to construct a mental representation. The goal of the research proposed here is to develop a model for parallel program comprehension that is based on the abstract mental representations formed by parallel programmers during program comprehension.

3. Research Ideas

Parallel programming has introduced new challenges including bugs that are hard to detect, making it difficult for programmers to verify correctness of code. One type of bug that occurs in parallel programming is data races. Data races occur when multiple threads of execution access the same memory location without controlling the order of the accesses and at least one of the memory accesses is a write (Liao et al., 2017). Depending on the order of the accesses some threads may read the memory location before the write and others may read the memory location after the write. Data races are difficult to detect and verify as they will not appear every time that the program is executed. To detect data races programmers must understand how a program executes in parallel on the machine and the memory model of the programming language.

In the proposed study, participants will be assigned the task of determining if a parallel program contains a data race. Participants will then be asked to identify the location of the data race if they believe that one exists and their level of confidence in their answer. The task of searching for a data race will be used to stimulate the comprehension process and as a result the programmer will form an abstract mental representation of the program. To determine if a program contains a data race programmers must mentally execute the program. This requires understanding how the program would execute on the machine and the possible interaction of executing multiple timelines of the program in parallel. To study the comprehension process an eye tracker will be used. Data from the eye tracker that may be able to assist in creating a model of program comprehension would be the order the code is read in, the sections of code that are revisited, and how often and how long the programmer spends reading sections of the code. This information will help to reconstruct their process for understanding and inform how they model the code. Additional information will be collected from participants in the form of questionnaires that will be developed to gain insight into their mental representations and understanding of the code.

Participants will be expert parallel programmers. The study of experts is important for informing the development of programming languages, instructional practices, and tools. To perform research on expert programmers it is necessary to be able to determine if participants are in fact experts. There has been a lack of agreement among researchers on how expertise should be measured and as a result there remains no standard for measuring programmer expertise. The distinction between expert and experienced also needs to be established. Experience is a measure of time spent working in a particular field or performing a task but does not necessarily translate into expertise, which is a measure of performance (Ericsson et al., 2006). Another research activity is to develop a tool for assessing programmer expertise.

4. Conclusion

Detecting the presence of data races in parallel code requires understanding the memory model of the programming language and how the program executes in parallel at the machine level. Using this task to stimulate the comprehension process will likely result in the formation of abstract mental representations with additional dimensions compared to those formed during the comprehension of sequential code. Through eye tracking and questionnaires we will develop a model for the mental representations formed by expert parallel programmers. To determine the level of expertise of participants, criteria will need to be developed that evaluates their programming skills.

5. References

- Détienne, F. (2001). *Software design-cognitive aspect*. Springer Science & Business Media.
- Ebcioğlu, K., Sarkar, V., El-Ghazawi, T., Urbanic, J., & Center, P. S. (2006). An experiment in measuring the productivity of three parallel programming languages. In *Proceedings of the third workshop on productivity and performance in high-end computing* (pp. 30–36).
- Ericsson, K., Charness, N. E., Feltovich, P. J., & Hoffman, R. R. (2006). *The cambridge handbook of expertise and expert performance*. Cambridge University Press. doi: 10.1017/CBO9780511816796
- Hochstein, L., Carver, J., Shull, F., Asgari, S., Basili, V., Hollingsworth, J. K., & Zelkowitz, M. V. (2005, 11). Parallel programmer productivity: A case study of novice parallel programmers. In *Supercomputing, 2005. proceedings of the acm/ieee sc 2005 conference* (p. 35–35). doi: 10.1109/SC.2005.53
- Liao, C., Lin, P.-H., Asplund, J., Schordan, M., & Karlin, I. (2017). Dataracebench: A benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the international conference for high performance computing, networking, storage and analysis* (p. 11:1–11:14). ACM. (event-place: Denver, Colorado) doi: 10.1145/3126908.3126958
- Mattson, T., & Wrinn, M. (2008). Parallel programming: Can we please get it right this time? In *Proceedings of the 45th annual design automation conference* (pp. 7–11). New York, NY, USA: ACM. doi: 10.1145/1391469.1391474
- Pennington, N. (1987). Empirical studies of programmers: Second workshop. In G. M. Olson, S. Shepard, & E. Soloway (Eds.), (pp. 100–113). Norwood, NJ, USA: Ablex Publishing Corp.

Challenging users' perceptions of decision boundaries in machine learning systems (WIP)

Rob Bowman

robert.bowman@cantab.net

Luke Church

Computer Laboratory

University of Cambridge

luke@church.name

Abstract

We investigate the design of smart systems that drive the exploration of decision boundaries created by Machine Learning (ML). We are interested in users' perceptions of decision boundaries, particularly in scenarios that involve the categorisation of non-categorical data. We present a system for colour categorisation using active learning with decision boundary visualisation and functionality for investigating users' perceptions of the decision boundaries they are teaching the system.

1. Introduction

One of the ways of writing programs is providing examples. This was studied early on as 'programming by example' (Lieberman 2001). The broad adoption of statistical machine learning techniques has changed this from an important, but niche, research question to a field with broad social implications.

Historically, it has often been asserted that Excel was the most popularly used programming language (Scaffidi et al 2005), this has now been displaced by labelling of machine learning systems through everyday interaction with ML systems such as Google Search, Spam/No-spam filters or other every day labelling tasks.

However, labelling is subject to a wide set of biases, and is an active area of research (Blackwell 2018). In this paper we describe a work in progress to build a tool for probing labelling behaviour in uncertain cases, with the intention of using it as a tool for providing creative interventions in tools used by artists.

2. Approach

2.1. Colour Categorisation System

We are implementing a prototype system that will be used in user studies to capture empirical results.

The premise of our system is a colour categorisation task. The system is initialised with points from the HSL (Hue, Saturation, Lightness) colour space with the objective of classifying each point as 'blue', 'green' or 'neither'. The motivation for our choice of task is that it requires the user to categorise non-categorical data, there is existing work which uses this task to study the phenomena of categorical perception that we can draw on, and we anticipate that it will lend itself well to decision boundary visualisation.

We frame the task as an active learning task. Active learning is a form of machine learning where the machine learning system learns to label an unlabelled dataset through interactively querying an information source such as a human user. Active learning is typically applied in situations where unlabelled data is abundant and manual labeling is expensive. A key challenge of active learning is the selection of the next data point to query the label of. There are a wide range of algorithms for this, the choice of which depends on the objective of labeling the system is learning.

In our active learning system the unlabelled data is a set of HSL points and the objective of the system is to learn how to classify each point through querying a user to label points. Our system uses support-vector machines to learn a probabilistic classification of the dataset. We select the next point

for labelling by computing the point with minimum difference in belief between its two most likely classifications. The support vector machine will learn decision boundaries which are partitions of the input space where it will classify points on either side differently.

We expect decision boundaries to form partitioning ‘blue’ from ‘neither’, ‘green’ from ‘neither’ and ‘green’ from ‘blue’ during experiments. We will analyse where decision boundaries form to study if biases occur, if the decision boundary visualizations can decrease bias in the learning process and the effects of factors such as the order of data points during labelling.

2.2. Research question 1: Do decision boundary visualizations improve users perceptions of machine learning models?

During uses of this system we are interested in users’ perceptions of what the machine learning component is doing. We are interested in how users expect the ML system to label unlabelled points at different stages during an experiment:

- Before any points have been labelled
- When a small number of points have been labelled
- When a large number of points have been labelled

The system will be able to query the user to capture their belief as to how the system will classify new points which can then be compared with the ML systems actual behaviour to record measure of the error in users’ perceptions.

We will record this error under the system in two modes.

1. A minimal system which presents the user with a visualization of only the target colour to label or predict the systems labelling of.
2. A system augmented with decision boundary visualizations.

We will look to address the research question by contrasting the recorded perception error under these modes.

2.3. Research question 2: Can decision boundary visualizations be used to reduce the effects of hysteresis?

We are interested by hysteresis effects whereby the users perception of how a datapoint should be labeled is affected by the preceding interactions with the system.

An approach we will take to exploring if there are such effects in this system is to use a system mode that queries the user to label colours in a sequence that transitions across a decision boundary. We will investigate how this impacts the positions of decision boundaries and users’ perceptions of what the system is learning.

3. Current Status

We have implemented a system that enables the querying of users’ to label points and express a belief about how the system would label a point. The system uses support vector machines to learn a classification over the colour space. We are currently considering options for the visualisation of decision boundaries. Figure 1 shows a screenshot of the interface in a labelling mode. The upper graphic within Figure 1 shows a decision boundary visualisation.

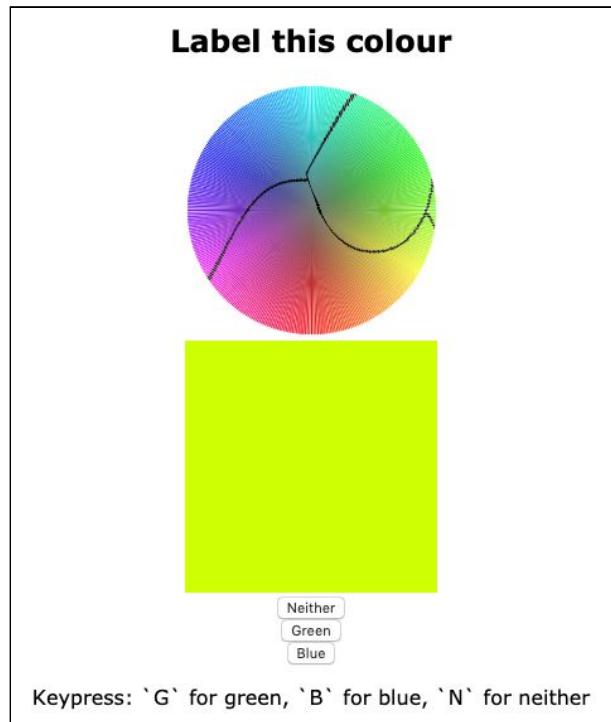


Figure 1 - Screen shot of the current work-in-progress system showing the labelling mode with decision boundary visualisation.

The system provides us with the functionality to run a range of experiments that are characterised by different sequences of interactions.

4. Forthcoming Work

We will be performing pilot studies test the setup of our experiment and options for decision boundary visualisation. We will then perform larger scale user studies to collect the empirical results which we will analyse in the context of our research questions.

5. References

- Blackwell, A. F. (2018). Interaction with Machine Learning.
<https://www.cl.cam.ac.uk/teaching/1819/P230/>
- Lieberman, H. (2001). Your Wish is My Command: Programming by Example. Morgan Kaufmann.
- Scaffidi, C., Shaw, M., and Myers, B. (2005) Estimating the numbers of end users and end user programmers. In Proceedings of *IEEE Visual Languages and Human-Centric Computing (VL/HCC)* 2005.

Probes and Sensors: The Design of Feedback Loops for Usability Improvements

Luke Church

Computer Laboratory
University of Cambridge
luke@church.name

Emma Söderberg

Department of Computer Science
Lund University
emma.soderberg@cs.lth.se

Abstract

The importance of user-centric design methods in the design of programming tools is now well accepted. These methods depend on creating a feedback loop between the designers and their users, providing data about developers, their needs and behaviour gathered through various means. These include controlled experiments, field observations, as well as analytical frameworks. However, whilst there have been a number of experiments detailed, quantitative data is rarely used as part of the design process. Part of the reason for this might be that such feedback loops are hard to design and use in practice. Still, we believe there is potential in this approach and opportunities in gathering this kind of ‘big data’. In this paper, we sketch a framework for reasoning about these feedback loops - when data gathering may make sense and for how to incorporate the results of such data gathering into the programming tool design process. We illustrate how to use the framework on two case studies and outline some of the challenges in instrumentation and in knowing when and how to act on signals.

Keywords: user-centred design, feedback, instrumentation, adaptive systems

1. Introduction

Programming changes over the lifetime of a system or language. At PPIG in 2018 we described early work in the diachronic study of notations (Church et al, 2018), especially of programming; that is, looking at how notations evolve overtime. However, it’s not only the notations that change, the interfaces used for programming change more rapidly, and the APIs and frameworks even more rapidly still.

It’s no longer necessary within most of the programming tooling community to argue for a user centric approach to these changes. Thanks to advocacy, including from the PPIG community, we know that we need to understand developers in order to motivate improvements to languages and their associated tooling.

But where does this understanding come from? Some of it comes from controlled experiments e.g. (Sime, Green, & Guest, 1973), some from field observations e.g. (Nardi, 1993), some of it comes from analytical frameworks e.g. (Blackwell, 2002). Despite a number of efforts, such as (Brown, Kölling, McCall, & Utting, 2014), comparatively little of it comes from empirical data from instrumentation in tooling.

This seems strange - it is, if anything, the era of ‘big data’, or as the advocates of the application of statistical techniques to improvement development call it ‘big code’. In the past we’ve written about possible strategies for applying machine learning to development (Church, 2005), (Church, Söderberg, Bracha, & Tanimoto, 2016). These techniques relied on tools for gathering substantial amounts of data about developers. If we can build systems that use statistical aggregations of data to create adaptivity it should be possible to use data to inform design work on the iteration of our tools? We continue to see this as an opportunity.

This challenge of making this work effective, was succinctly summarized in (Lewis, 2016), where Clayton suggests that ‘measurement is considered hopeless’ due to the varied and intersectional nature of the design of programming context. Here we explore an alternative hypothesis - that it isn’t always hopeless, but it works in some places and not in others. In this paper we aim to describe some of those conditions.

We sketch a framework and use it to characterise a number of previous instrumentation systems built to support design feedback. We then use this framework to exemplify where instrumentation works and where it doesn’t and to share the lessons we’ve learnt along the way.

We’ll start by identifying the decisions that need to be made, both in the instrumentation system and in the larger feedback loop.

2. Framework: Sensors, Probes, and Feedback

2.1 What to Measure

In this paper we will separate the discussion into the instrumentation system, the technical component that senses information and records it in some manner, and the broader socio-technical feedback system that it is embedded within that includes decision making processes based on the feedback.

These two components have a chicken-and-egg relationship with each other. The design of the instrumentation system is affected by the types of feedback that can be operationalised, and the types of feedback that can be operationalised are determined by what can be measured.

There are two broad strategies that get taken: record everything in anticipation of unknown questions, and try and make sense of it later, and record information about specific questions in the knowledge that you will only want to answer that question.

These strategies also determine how much interaction we expect to have with users. Some data-capture systems operate completely autonomously, not interacting with users during the capturing of data, others require specific interaction from their users. Broadly we refer to these instruments as sensors (autonomous, no interaction with the user), and probes (have specific interactions). We’ll elaborate on these categories more shortly. We are not suggesting that either of these strategies is ‘correct’ - that depends on the circumstances.

2.2 Lengths of the Feedback Loop

There are a number of different things that can be done with measurements. At the tightest interaction loop, they could provide data that drives a self-adaptive system (Brun et al., 2009), responding immediately to the sensed data to adjust the interface and change what the next keystroke will do, for example like the smart text entry systems on mobile phones.

At completely the opposite end of the reactivity spectrum, the sensors might provide data that inform the next generation of programming language, often a matter of years after the data has been collected. The data can also be used summatively to assess how well something worked (did users accept the quick fixes the IDE offered?), or formatively to guide the design of the next generation of a system (Where do the majority of errors in code come from?).

2.3 Known and Unknown Problems

The final inter-tangled axis is whether the endeavour is to address a known problem, or an unknown one. For example, a known problem - and one we’ll discuss shortly - is that static analysis systems have false positive rates. An unknown problem might be closer to ‘improve the syntax of this language’. Of course this is not really a dichotomy, but rather a spectrum of how well pre-characterised the problem is.

2.4 How to Measure?

There are different places in which we could do the measurement. We characterise this by considering the ‘distance’ from the user. We could measure something about the user themselves, about the users’ interaction with the computer, or about just the computer. The latter are studied by the systems communities, with distributed logging tools such as Kafka¹ or commercial products like StackDriver². In this paper we’ll focus on the first two cases, measuring programmers and their interaction.

2.4.1 Biometrics: Measuring the Programmer

Biometric sensors offer an opportunity to capture signals enabling detection of, for instance, task difficulty (Fritz, Begel, Müller, Yigit-Elliott, & Züger, 2014), programmer proficiency (Busjahn et al., 2015), and developer frustration (Müller & Fritz, 2015). There are a number of devices measuring biometric signals available, for instance, eye-trackers, electroencephalography (EGG) headbands, and wristbands measuring properties like temperature and blood volume. Data from such devices can be used on their own or in combination with each other to enable more complex detection.

In this setting, we are considering biometric sensors as passive collectors of information and not as devices for control. Especially eye-trackers are being explored as control devices in systems like EyeDE - eye-tracker controlled actions in an IDE (Glücker, Raab, Echtler, & Wolff, 2014), EyeNav - code navigation with eye movement (Radevski, Hata, & Matsumoto, 2016), and CodeGazer - eye-tracker controlled code navigation (Shakil, Lutteroth, & Weber, 2019). Sensors being used in this way have other, but sometimes related, kinds of challenges. For instance, eye-trackers have to deal with the so called “midas touch” issue, when gaze-based functionality is triggered without user consent.

Even though these sensors do not actively prompt a user for information they do incur other user costs; a user may have to wear the measuring device, or calibrate a device before it can be used. The user cost in this regard has been reduced in recent years with less intrusive free-standing eye-trackers³, and non-sticky EEG measuring devices in the form of headbands⁴. Still, the user needs to commit to an additional device in their environment or on their person. The device needs to be considered useful enough to overcome that cost.

2.4.2 Instrumentation: Measuring the Interaction

Behavioural instrumentation systems are pieces of software running on the computer that a developer is using. The software observes one or more activities that the developer performs, and reports it to a server for archiving and analysis.

For example we have previously used this to study code completion behaviours (Mărăşoiu, Church, & Blackwell, 2015). Others have aimed to build general purpose instrumentations similar to the platform used in Dynamo above. BlackBox (Brown et al., 2014) is an example of one such system used in the educational platform BlueJ.

Sensors like this offer the opportunity of capturing a very wide range of data for subsequent analysis. As demonstrated in the Dynamo case, they can be useful for determining behaviours which are not known at the time when the system is being originally designed.

A full discussion of the technical implementation strategy is beyond the scope of this paper, however we have learnt several lessons by building and operating a number of these tools, two of which are important for the rest of the discussion.

¹ <https://kafka.apache.org/>

² <https://cloud.google.com/stackdriver/>

³ <https://www.tobii.com/>

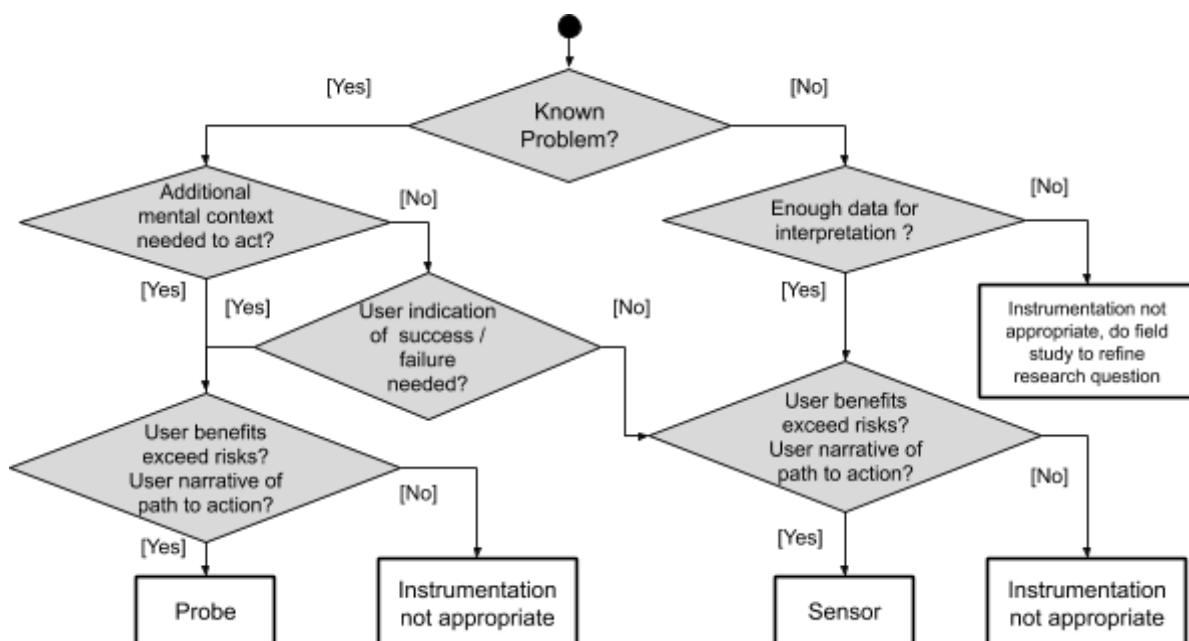
⁴ <https://brainbit.com/>

Firstly: It is economically feasible to build instrumentation systems that respond to user actions, but often not to system actions. For example, the first author built a system that recorded code completion requests within Eclipse (triggered by a user pressing '.'), however when the system was extended to record internal exceptions thrown by Eclipse (triggered by a system event), the data volume overwhelmed the infrastructure. This limits what can be effectively recorded.

Secondly: It is often very useful in analysis to be able to, as much as possible, recreate the context that the user was in when they performed an action. For example, what source code where they looking at? In order to build systems that are practically resilient to data loss problems such as network failure, it helps to send either all the information with each request, or to send periodic ‘keyframes’ from which

2.4 The Framework

With this structure, of known and unknown problems, opportunities for measurement, risks and returns and probes and sensors, we can sketch a decision process for deciding when and in what form to use instrumentation.



We've outlined a framework, now we're going to show you how to use it by applying it to two examples: Tricorder and Dynamo.

4 Case Studies

4.1 Tricorder

Critique

CL 12345

//path/to/file/Test.java

Reviewer Please add more tests

[Reply](#) [Reply with quote](#) [Done](#) [Ack](#)

```
package path.to.file;

public class Test {
    public boolean foo() {
        return new String("foo") == "foo".toString();
    }
}
```

package path.to.file;
/**
 * Sample class
 */
public class Test {
 // Sample method
 public boolean Foo() {
 return new String("foo") == "foo".toString();
 }
}

Lint warning: Name 'Foo' must match pattern

[Not useful](#)

Figure 1: Example of a robot comment displayed on a line on a code review diff. The ‘not useful’ probe is integrated as a link in the bottom right corner.

We describe this case in terms of the system, the known problem, the probe, and the collected signal:

The System: As an example, we will consider Tricorder (Sadowski et al. 2015), a system running internally at Google which integrates results from program analyzers into tools used by software developers. The most prominent integration point is the code review tool Critique, where the system presents analysis results in the form of robot comments on code under review. That is, next to comments made by humans on diffs of code, comments from program analyzers are presented using a different color - grey is used for robot comments and yellow for human comments (as shown in Figure 1).

The Known Problem: False Positives: Use of program analysis tools is plagued with a history of problems with false positives (Johnson, Song, Murphy-Hill, & Bowdidge, 2013) causing users to not trust the results and eventually ignore them (Imtiaz, Rahman, Farhana, & Williams, 2019). An awareness of this problem has been central to the shaping of the philosophy behind the Tricorder system (Sadowski, van Gogh, Jaspan, Söderberg, & Winter, 2015), which emphasizes data-driven usability improvements and the collection of ‘not useful’ feedback for analysis results.

The Probe: ‘Not useful’ Links on Robot Comments: When users are presented with program analysis results as robot comments during code review, they are presented with the option of clicking a link named ‘not useful’, placed on each individual robot comment, shown in Figure 1. If a user decides to click the link, a non-obstructing, e.g., not in the middle of their vision, notification pops up asking if they would like to provide more details. If a user decides to do so, they click a link and are then directed to a partially filled in bug template on a bug filing page.

The Signal: The What and Why of a ‘Not useful’ Analysis Result: In collection of ‘not useful’ clicks, information of the analyzer category not found useful is collected, and if the user decides to

provide more information, details of the position of the analysis result in the code under review is collected and inserted into the bug template presented to the user.

The ‘what’ of the signal is thus collected in two steps, first just the analyzer category, for instance, a specific bug pattern detected by ErrorProne (“Error Prone,” n.d.), and second details of the code context in which the analyzer presented a result, that is, the version of the file under review and the position of the comment plus the message provided to the user. The ‘why’ of the signal is potentially answered in the second step where users provide a comment in the bug template to explain why the result was not found useful to them.

The second step also provides a forum to have a discussion about the reasoning behind the bug report to further investigate why the report was filed. In the Tricorder system this second step has been very useful in understanding why results are not found useful. An analyzer category may, for instance, get a lot of bugs filed against it due to incomprehensible messages in the analyzer result (Sadowski et al., 2015).

With numerous analyzer improvements due to the ‘not useful’ signal in Tricorder, it appears to be a well-chosen and successful signal. What can we learn from this example? Why does this signal work?

4.1.1 Discussion

Analyzing the ‘not useful’ signal with the properties from Section 2 in mind, we hypothesize as follows:

Activity/Elements That the cost of collecting feedback is lowered by using common at hand elements in the domain. In the context of code review, a domain where adding, reading, and replying to comments is the main activity and using mouse clicks to navigate, at least among comments, the collection cost is lowered by using elements like comments and clickable links. As a comparison, imagine a scenario where the user would be encouraged to leave feedback, but would be sent to a draft of an email program or a spreadsheet instead to manually enter ‘not useful’ click feedback.

Activity/Mindset That the mindset of the reviewer and reviewee in code review present especially fertile ground for explicit and detailed signal collection. Users are in a mode of inspection with the goal of scrutinising and improving the code under review (Sadowski, Söderberg, Church, Sipko, & Bacchelli, 2018). In this state of mind, with fresh mental models of the code in their mind ready for inspection, the effort for providing detailed feedback to an analysis result perceived as not useful is lower. For comparison, imagine being presented with a snippet of code outside code review together with an analysis result and then having to load in the mental models of the code to perceive the usefulness of the analysis results and to potentially provide an explanation as to why it is not useful.

Organization/Culture That the culture among the users of Tricorder, that is, among Google software engineers, encourages feedback (“If something is broken, fix it”) (“Alphabet Investor Relations,” n.d.). A cultural motivation may provide a constant micro encouragement to react to any annoyance and click the link. As opposed to a culture where filing a bug would be frowned upon or even considered offensive.

System/Trust That users experience that the effort of filing a bug matters (return on investment, effort matters). The system needs to be responsive and act on the feedback. In the case of Tricorder, responsiveness is part of the philosophy in the analyzer contract, problems need to be addressed or an analyzer may be turned off. Unresponsive behavior for filed bugs would be one way of not following this usability contract. In addition to trust in the system, the system administrators can assume that there is no malicious intent from users within the same organization. In contrast, imagine having a system open to all users of the Internet.

Probe/Granularity That the probe is at the right granularity in terms of precision. The ‘not useful’ signal could probably be divided into different kinds of not usefulness, for instance, ‘I don’t understand’ or ‘this is incorrect’. However, we argue that this would require the user to make another

decision, that is, the user just decide to give feedback and now she also needs to decide which kind of feedback to give. The cost now becomes higher. To allow users to quickly provide a ‘not useful’ click, even if it means more than one thing, is a good trade off. This quick and perhaps coarse activity acts as a funnel, indicating usability problems and potentially also providing some answers to why for the cases where users decide to take the extra cost of filing a bug.

4.2 Dynamo

The System: Dynamo (“Dynamo BIM,” n.d.) is an open source hybrid textual/visual programming language, often used by architects in the design of buildings.

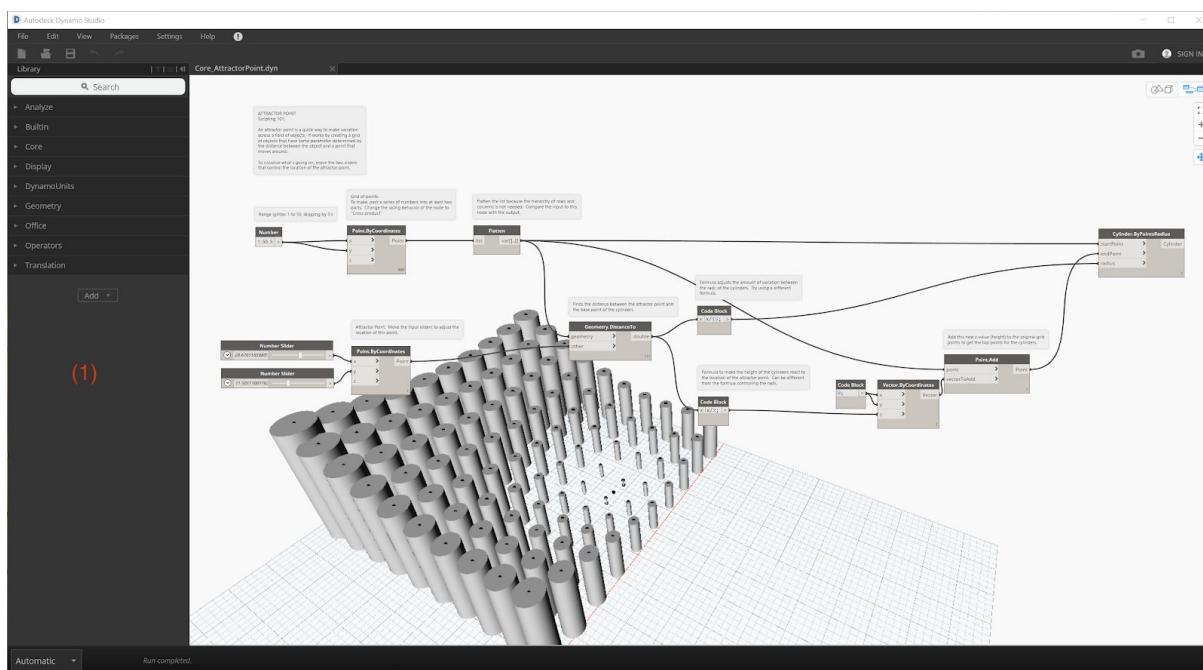


Figure 2: Dynamo in use create a cut out that might be used to pattern a facade on the side of a building.

The main Dynamo user interface, shown in Figure 2, consists of a canvas where ‘nodes’ are placed and linked together to make a graph. These nodes either directly represent expressions and function calls, or they contain short blocks of scripts where multiple expressions and function calls can be written together. Together these nodes make up a ‘workspace’. This workspace is the program that is executed. The result of executing that program is typically geometric and appears underneath the nodes.

Nodes get added to the canvas by the bar on the left labelled (1). This bar contains a hierarchical categorisation of the nodes by group. For example nodes to do with geometry are grouped together. Clicking on an element in the bar then adds it to the canvas. It also includes a mechanism for searching within the menu. For example typing in ‘point’ searches for items related to point, resulting in Figure 3:

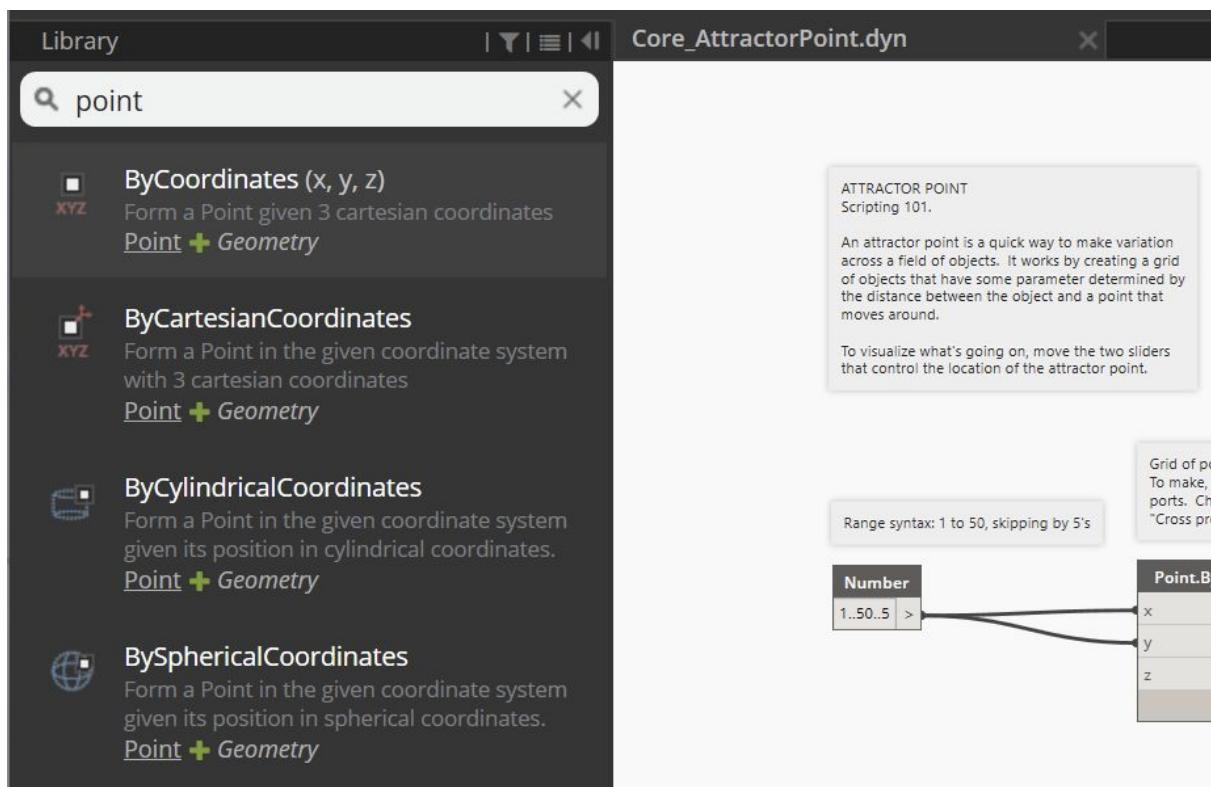


Figure 3: Dynamo's search window after searching for 'Point'

The Known Problem: Searching for APIs The design of the API for a system like Dynamo is hard. In the history of the Dynamo project a number of attempts had been to standardise the vocabulary in use in the API, however there are still a large number of cases where synonyms remain. For example, a user might think of making ‘a cube by extruding a rectangle’, or they might think of ‘creating a solid from points’. They will need to use the search tool, in connection with other tools like documentation to find the API they want to use. This relates closely to the problem of using code completion functionality in a traditional IDE (Mărăşoiu et al., 2015)

The Sensor: Search Results: To address the search problem a sensor was added to the search mechanism. It records what users [who have opted-in] type into the search bar, associated timing information, and whether or not they add a node to the canvas from the search window.

The Signal: Search Effectiveness By using a combination of character-to-character editing models, and timing analysis, we can determine which queries lead to the successful addition of a node into the canvas, and what synonyms the users were trying on their way to getting there. From this we can define ‘problem nodes’ - that is nodes which users need to perform multiple searches to get to, and ‘problem queries’ - that is queries that don’t usually result in a node being added. These can then be mapped by occurrence, so we can produce characterisation of severity and frequency of issues. The qualitative nature of the search queries helps in interpretation of what developers were looking for.

The [Mostly] Unknown Problem: Visual Languages and Usability As well as this specific problem of searching for APIs there is the more general concern. Visual programming languages are known to have significant usability difficulties (Green, Petre, & Bellamy, 1991). These issues were very well known to the team who built Dynamo, including the first author. We wanted to be able to empirically investigate the significance of these usability problems during the lifetime of the Dynamo project. This is of course a rather vague characterisation of a problem - but often one that design teams at the start of a practical project will be faced with.

The Sensor: Workspace Sampling To address the problem of the unknown usability improvement needs, Dynamo also contains a recording mechanism that takes a snapshot of the nodes and connectivity - but not the data flowing through them - in the workspace for users that have opted in. This snapshot is taken once per minute by using a mechanism that is derived from the file serialisation format.

The Signal: Dychronic Graphs The signal from the workspace sensor is a sample of the evolution of graphs over time. These can then be used to analyse the usage either synchronically, for example: ‘what is the most commonly used node?’ ‘in graphs which use nodes from Revit, are nodes from Dynamo’s native geometry library also used?’, or dycronically, for example “how is usage of Code Block Nodes changing?”

However, as is typical in purely sensor based applications - there is a lack of context associated with the data that can result in it being challenging to interpret. We know a lot about what users of Dynamo were doing, but can only infer why they were trying to do it from their actions. Because the information about what they were trying to do is only in their minds, there wouldn’t be any way of gathering this using a sensor - it would have needed a probe, and a fairly demanding one in terms of user time: explicitly asking the user what they were doing.

4.1.1 Discussion

Activity/Mindset As suggested above, the two cases here - search and workspace sampling have very different activities - and understanding of what is going on. The activity in search is fairly specific, the user is searching for a node to add to the canvas, whereas in general development it is unknown. This directly affects the utility of the analysis that is possible.

Utility: Both signals, of search effectiveness and graphs, have proven practically useful in providing a feedback mechanism from the users of Dynamo to the designers. The search effectiveness analysis has been used to both select synonyms to add to the search system as well as to evaluate proposed changes to the algorithm.

The workspace sampling has been used to evaluate the success of features (e.g. investigating how much impact the List at Level feature⁵ has had on the usage of an alternative feature that known usability difficulties (partial function application).

Interpretation: The lack of an explicit probe mechanism by which a user of Dynamo could indicate whether the behavior they have just experienced was one they had desired makes the data from both of these sensors more difficult to interpret. However, it does carry a minimum cost to the users.

The data from the search sensor has been considerably easier to interpret in practice, requiring only a few days of data scientist time to analyse into results that have been used to directly influence the design. The data from the workspace sampling system has been more difficult to interpret. The primary reason for this was the difficulty of understanding from the graphs what the developer was trying to do, and then turning that understanding into a query that could be applied automatically to a large number of graphs.

Missing Measurements: Despite the generality of the sensor for recording the workspaces, some data was missed, creating further challenges in the analysis. For example, a workspace can reference functions defined in other workspaces. If these were not loaded into the editor, they would not have been sampled by the sensor.

Sampling Bias Caused by Opt-in: The data that is recorded by the sensors is clearly sensitive, and requires explicit consent from the users. This is provided by the user checking an extra box agreeing to send the data. This process may result in a bias caused by, for example, users working on commercial projects not opting in to instrumentation.

⁵ <https://dynamobim.org/introducing-listlevel-working-with-lists-made-easier/>

5 Discussion

What can we learn from these two examples? The addition of empirical feedback loops has been useful in both cases, with the purpose ranging from disabling naughty analysers that cry wolf too often, to fixing discoverability issues in locating some nodes.

However, we can also see that the viability of a particular solution is dependent on the circumstance. The use of Critique is part of a highly normalised workflow, which means that it is possible to automatically store and reproduce all the context of a code review. This helps for looking at being able to empathise with the user of the tool. Usage of the Dynamo canvas is much less normalised, as it is used by a wide variety of different organisations for different tasks, so it's harder to know the context just from the data that has been recorded.

The change in the interpretability of the context is not the only thing that is contingent on the circumstances; so is the motivation to engage with the system probe or sensor in the first place. The code review system rests on the users taking the time to interact with the system, in the knowledge that they are doing so in a context where the results of their interaction will be looked at, and so a broader community will benefit. As a single organisation, that social context can be explicitly built by Google (see for example the code of conduct ("Alphabet Investor Relations," n.d.), a similar community spirit exists within users of Dynamo and constructing it has been an important part of design discussion (Kron, Personal communications)

Contextualising Trust

However as we can see this trust is contextual - the Tricorder system and its integration with the Critique code review system are both built and used by Google software engineers - consequently the developers using it feel assured that the incentives align effectively to making sure that their data will be held securely, as the same organisation that builds the system would be directly harmed by any problems that arose. The Dynamo case is considerably more complex, the majority of its users will not be employed by the same organisation that provided the infrastructure. Whilst there have been no security incidents associated with the system, the trust relationship is considerably different.

In the Tricorder/Critique context, the user trusts that their time is not being wasted; the data they provide will in fact be used to benefit themselves and a broader community. In other words, they trust that the system will behave as intended and will do what the user has requested. In the Dynamo case, the user doesn't pay any time cost after they have opted-in, however they trust that the data will be held and used responsibly.

Meaningful engagement with the system comes down to a couple of properties, firstly trust that the data will be held and used appropriately, however more subtly a trust that the data will be actioned so that the risk (of IP exposure, abuse use of data etc. in sensors, or of the above and time wasting for probes) is worth the engagement.

The key questions for each case then is how to interpret the information to support a feedback loop, how to take action on the basis of that information.

Interpretation and Action

Understanding the signal from a sensor or a probe is not usually trivial. The data in itself often feels impoverished to look at, as it's missing the extra context that you would have in an in-person study - notably asking the user what it was they were doing!

In the absence of these direct observations, it's tempting to try and use metrics to determine use. For example, it's common practice in technology development organisations to measure feature usages, similar to the usage signal from Dynamo above.

However even with a simple metric like usage, there are multiple possible causes that could result in an increased usage. It could be that the feature is serving user needs effectively, so people are choosing to use it more. Alternately it could be that the feature is very difficult to use, so users are needing to interact with it more in order to achieve the effect they need.

Another option that has an initial appeal is visualisation. If we could visualise the patterns of interaction, that might provide a way of understanding what was happening. We have used this technique before in (Mărăşoiu et al., 2015). However it doesn't solve the problem, whilst visualisation provides a mechanism for seeing highlighting distributional patterns, the structure of the visualisation still determines what can be seen and what can't.

As a heuristic, we have found that quantitative methods can be used effectively in cases where either (1) they are used in a mixed-methods approach to determine how much a phenomenon that has been observed generalises, or (2) where they are coupled to a specific action that the designer can take on the basis of the feedback, such as disabling a misbehaving analyser.

Self-adaptive Systems

One fairly extreme design strategy is to build *self-adaptive systems*, rather than waiting for the socio-technical processes to take place, where a researcher looks at the results and makes a design, the system adapts by itself. Systems that otherwise would be open-loop (involving a person in the design loop) are designed to be adaptive by incorporating closed-loop feedback to enable automatic adaptation (Brun et al., 2009) without a person involved. The general reasons for development of self-adaptive systems is to make it manageable to deal with a changing environment, changing requirements, and uncertainty. The design of the feedback loop which is central to the design of the system.

This centrality of the feedback mechanism is shared with the framework outlined in this paper. In the framework, there should be a narrative to the user connecting the action that may be the result of feedback they provide, and in an self-adaptive system the feedback loop, or control loop, should be known to the user (Brun et al., 2009).

The majority of self-adaptive systems are not end-user applications, but rather systems controlling, for instance, the numbers of VMs spun up for a cloud application to handle different kinds of load. There has been some early exploration of self-adaptive interfaces (Innocent, 1982) and examples can be found in, for instance, adaptive text input applications in mobile phones.

We're not aware of examples of systems of closed-loop self-adaptive systems for use by developers. Many years ago we experimented with using the Dasher interface to build a self-adaptive system based on a statistical model for text entry (Church, 2005). It had an awkward tendency to go into what would now be described as 'extreme over-fitting', paying rather too much attention to what the developer did last and repeating that as suggested code.

Whilst the system was unbearable to develop with, it at least suggested that there is an alternative counterpoint; a feedback cycle to the developer that was too close and too obvious!

6 Conclusions

We started this paper discussing Clayton's worry that "measurement was hopeless". Much of his concern was that the variation in the users, and the reasons why they are programming, create variation that swamps the quantitative measurements that can be taken. This, combined with the problem that the quantitative methods don't give data that can be used to guide formative processes makes measurements in controlled experiments a very difficult approach to take to build design.

This paper represents a slightly alternative perspective. Whilst we agree with many of Clayton's critiques in the context of controlled experiments, we suggest that automatic measurement (sensors)

and semi-automatic measurement (probes) can indeed be turned into useful feedback mechanisms to drive design.

They offer the ability to measure users in the wild, as they go about doing their wide range of varied tasks, so have strong external validity. These tools can scale to measure entire populations - making substantial inroads to the problems of tracking variation. By careful design and analysis we've shown examples where measurements have been practically useful.

They are by no means a panacea. We've shown that involvement in these systems from the users' perspective entails some risk - from having their time wasted, to security and perspective risks. It's easy to see how users can become rightly sceptical about doing this in exchange for platitudes about product improvement, they expect something concrete in return, or at the very least a concrete narrative about how the designer of the system is proposing to use the data to get there.

Looking ahead, these feedback loops in non self-adaptive systems aren't obvious. This is the case in product development, but is also the case in research. How will taking part in a study on C++ help make someone's future better?

As a community we've accepted that understanding users is needed, but there's future work to do in improving how we talk about how we'll use the feedback loops that we're creating both in our experiments and in our instrumentation systems to improve developers' experiences.

References

- Alphabet Investor Relations. (n.d.). Retrieved June 28, 2019, from Alphabet Investor Relations website: <https://abc.xyz/investor/other/google-code-of-conduct>
- Blackwell, A. F. (2002). First steps in programming: a rationale for attention investment models. *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, 2–10.
- Brown, N. C. C., Kölling, M., McCall, D., & Utting, I. (2014). Blackbox: a large scale repository of novice programmers' activity. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 223–228. ACM.
- Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., ... Shaw, M. (2009). Engineering Self-Adaptive Systems through Feedback Loops. In B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, & J. Magee (Eds.), *Software Engineering for Self-Adaptive Systems* (pp. 48–70). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., ... Tamm, S. (2015). Eye Movements in Code Reading: Relaxing the Linear Order. *2015 IEEE 23rd International Conference on Program Comprehension*, 255–265.
- Church, L. (2005). *Introducing #Dasher, A continuous gesture IDE, A work in progress paper*. Retrieved from <http://www.ppig.org/sites/ppig.org/files/2005-PPIG-17th-church.pdf>
- Church, L. Marasoiu, M (2018): *A growing tip or a sprawling vine: How software grows*.
- Church, L., Söderberg, E., Bracha, G., & Tanimoto, S. (2016). *Liveness becomes Entelechy - A scheme for L6*. Retrieved from <https://ai.google/research/pubs/pub45596>
- Dynamo BIM. (n.d.). Retrieved June 28, 2019, from <https://dynamobim.org>

Error Prone. (n.d.). Retrieved June 28, 2019, from <http://errorprone.info/>

Fritz, T., Begel, A., Müller, S. C., Yigit-Elliott, S., & Züger, M. (2014). Using Psycho-physiological Measures to Assess Task Difficulty in Software Development. *Proceedings of the 36th International Conference on Software Engineering*, 402–413. New York, NY, USA: ACM.

Glücker, H., Raab, F., Echtler, F., & Wolff, C. (2014). EyeDE: gaze-enhanced software development environments. *Proceedings of the Extended Abstracts of the 32nd Annual ACM Conference on Human Factors in Computing Systems*, 1555–1560. ACM.

Green, T. R. G., Petre, M., & Bellamy, R. K. E. (1991). *Comprehensibility of visual and textual programs: A test of superlativism against the “match-mismatch” conjecture*. Retrieved from <http://dx.doi.org/>

Imtiaz, N., Rahman, A., Farhana, E., & Williams, L. (2019). Challenges with Responding to Static Analysis Tool Alerts. *Proceedings of the 16th International Conference on Mining Software Repositories*, 245–249. Piscataway, NJ, USA: IEEE Press.

Innocent, P. R. (1982). Towards self-adaptive interface systems. *International Journal of Man-Machine Studies*, 16(3), 287–299.

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why Don'T Software Developers Use Static Analysis Tools to Find Bugs? *Proceedings of the 2013 International Conference on Software Engineering*, 672–681. Piscataway, NJ, USA: IEEE Press.

Lewis, C. (n.d.). *Methods in user oriented design of programming languages*. Retrieved from <http://www.ppig.org/sites/ppig.org/files/2017-PPIG-28th-lewis.pdf>

Mărăşoiu, M., Church, L., & Blackwell, A. (2015). *An empirical investigation of code completion usage by professional software developers*. Retrieved from <http://www.ppig.org/sites/default/files/2015-PPIG-26th-Marasoiu.pdf>

Müller, S. C., & Fritz, T. (2015). Stuck and Frustrated or in Flow and Happy: Sensing Developers' Emotions and Progress. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 1, 688–699.

Nardi, B. A. (1993). *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA, USA: MIT Press.

Radevski, S., Hata, H., & Matsumoto, K. (2016, October 23). *EyeNav: Gaze-Based Code Navigation*. <https://doi.org/10.1145/2971485.2996724>

Sadowski, C., Söderberg, E., Church, L., Sipko, M., & Bacchelli, A. (2018). *Modern Code Review: A Case Study at Google*. Retrieved from <https://ai.google/research/pubs/pub47025>

Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E., & Winter, C. (2015). Tricorder: Building a Program Analysis Ecosystem. *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, 598–608. Piscataway, NJ, USA: IEEE Press.

Shakil, A., Lutteroth, C., & Weber, G. (2019). CodeGazer: Making Code Navigation Easy and Natural

With Gaze Input. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 76:1–76:12. New York, NY, USA: ACM.

Sime, M. E., Green, T. R. G., & Guest, D. J. (1973). Psychological evaluation of two conditional constructions used in computer languages. *International Journal of Man-Machine Studies*, 5(1), 105–113.

Cognitive Dimensions of Modular Noise Improvisation

James Noble

Engineering & Computer Science
Victoria University of Wellington
kjx@ecs.vuw.ac.nz

Abstract

The availability, affordability, and portability of Eurorack format modular synthesizers has lead to an increase in their use in live, improvised performance. Decreasing prices and physical size, coupled with increasing reliability has meant modularity are finally leaving the studio and appearing on stage. While modular synthesizers are typically prepatched (configured at leisure) ahead of time, contemporary synthesists are adopting livepatching — wiring up a modular — as an integral part of their performance practice. This paper uses the cognitive dimensions framework to analyse the programmatic content of modular livepatching, in the context of the author's experience with modular synthesizers for performing improvised noise.

1. Introduction

Modular synthesizers are finally leaving the esoteric world of academic electronic music studios and moving into concert halls, rock venues, and dodgy experimental performance spaces (Paradiso, 2017; Auricchio & Borg, 2016). The key innovation has been the gradual adoption of the Eurorack format for modular synthesizers, developed by Doepfer in the late 1990s, which are physically smaller than classical modulars of the 1960s (Moog and Roland modules were 5U high and used large “1/4 inch” (6.35mm) plugs and sockets; Eurorack modules are 3U high and use 2.5mm plugs and sockets). Relying on electronics of the 1990s and beyond (rather than the 1960s) means Eurorack modulars are cheaper and lighter than traditional modulars (as well as smaller), greatly easing their practicality for musical and sound-art performance (Rossmy & Wiethoff, 2019).

This paper interprets modular synthesizer patching — particularly “livepatching” where modules are (re-)connected as part of a public noise-art performance¹ — as a live physically-embodied domain specific programming language (see Figure 1. Modular synthesizers are essentially domain specific analogue computers; configuring analogue computers by patching modules together and adjusting parameters is domain specific programming, distinct from “playing” the synthesizer by triggering notes from a keyboard. “Livepatching” — adding and changing connections between modules and adjusting the parameters of those modules — is the modular analogue of music performance by live programming — a.k.a. “livecoding” (McLean, Rohrhuber, & Collins, 2014; Hutchins, 2015).

The next section briefly overviews modular synthesis and gives the context of my experience livepatching improvised modular noise. Section 3 then



Figure 1 A Quiet Noise: Winter Livepatching

¹A Quiet Noise: Winter, rDc, Dunedin, 14 June 2019. Further illustrations and video available from <https://dunedinsound.com/gigs/a-quiet-noise-winter/> by fraser@dunedinsound.com under CC-BY-SA 4.0.

considers livepatching using the cognitive dimensions framework, section 4 discusses some related work, and section 5 concludes.

2. Livepatching Eurorack

For the last five years I've been livecoding, improvising, and livepatching as half of "Selective Yellow", an experimental improvisation duo of indeterminate orthography drawing on New Zealand's heritage of experimental music practice (Russell, 2012; McKinnon, 2011) that seeks to recreate (electronically) all the worst excesses of free jazz with all the enthusiasm of antisocial teenagers meeting their first MOS6851 (Wilson & Noble, 2014). Selective Yellow performances typically employ a number of different synthesizers or sound generators as well as modular synthesizers, ranging from digital toys (Kaoscillators, Buddhamachines) to semi-modular analogue and MIDI digital synthesizers, played with a variety of controllers (wind controllers, monome grids, knob boxes etc) — while eschewing a primary role for digital audio workstation software and computer-based virtual instruments. Since Selective Yellow rehearses and performs only a few times each year, we're probably only Grade 4 (Nilson, 2007).

Figure 1 shows the configuration of typical modular synthesizer — a smaller version of the modular I used in a recent Selective Yellow performance. The modular rack is in front of the performer, who is interacting with a separate control surface (Beatstep Pro sequencer). A couple of smaller standalone synthesizers are to either side of the main modular. There is a tangle of patch cables on the modular, some from the sequencer to the modular proper, the majority interconnecting modules in the rack.

Figure 2 shows a simple modular patch. On the left a MIDI module is providing a regular clock signal; this signal is being used to control the sequencer module next to it. The outputs from the sequencer (a gate signal and a pitch control voltage) are then used to control the Edges oscillator bank module. The "MIX" audio output of Edges is linked into the audio input of the Wasp filter; the cutoff frequency of the filter is modulated by two separate LFOs from the Quadruple LFO module. Finally the filter's output is patched into the "Outs" module, which can then be connected to a terrifyingly large amplifier.



Figure 2 Example modular patch. Screenshot of modularGrid.com. From left to right the modules are MIDI (as clock source); micro-sequencer; Edges oscillator bank; Wasp filter; Quadruple LFO; Outputs.

The dynamics of textual livecoding — performance programming to produce music — using relatively traditional textual programming languages and editors or development environments has been well examined (McLean et al., 2014; A. Blackwell, McLean, Noble, & Rohrhuber, 2014; Magnusson, 2014). One of the goals of Selective Yellow has been to transfer that aesthetic from the disembodied, virtual, digital world of programming languages to the embodied, tangible, analogue world of modular synthesizers.

sis. Our approach is similar to Hutchins (2015) in following classical livecoding practice (Nilson, 2007): as much as possible starting with an empty, unpatched synthesizer; performing the patching either where it can be seen directly by the audience (Figure 1 was taken from the audience less than two metres from the performers) or, in larger venues, indirectly projecting an image of the modular to the audience; and explicitly focusing on the programmatic aspects of patching — connecting modules via patch cords.

Modular livecoding is centred on two main activities “patching” and “tweaking” (Bjørn & Meyer, 2018). Patching is constructing circuits by connecting modules together, and tweaking (also “twiddling”) involves adjusting the settings of the patched modules by e.g. turning the knobs or operating other controls on their faceplates, or adjusting settings on control surfaces. Neither patching nor tweaking are really direct analogues of playing a traditional musical instrument: tweaking comes closest but is often more tentative or exploratory.

Modular synthesizers can also include traditional performance-oriented controllers such as joysticks, light sensors, monome grids, or theremins, and via MIDI or OSC, can have access to essentially any contemporary digital performance controller. This gives rise to a third potential activity — that of an instrumental soloist where much of the expression of the generated sound is under the performer’s immediate control. (Even without such controls, there is always the option of performing a solo by “playing” the pitch control knob of an oscillator and filter cutoff or mixer attenuation in real time. I find this soloing activity qualitatively different to adjusting module parameters to configure their place in a larger patch: tweaking an oscillator pitch control while gurning like a 70s guitar hero is direct and immediate, much more like playing a conventional musical instrument, while configuring a module by turning a knob that controls the amount of LFO modulation to be applied to the pitch is much more indirect and delayed, and much more like programming.

3. Cognitive Dimensions

The *Cognitive Dimensions* framework (Green, 1989; Green & Petre, 1996) analyses notations of any kind, including but by no means limited to data or code visualisations and programming languages. For example, the framework has been used to compare and contrast Ableton Live and the ChucK programming language (A. F. Blackwell & Collins, 2005) and to evaluate the design of a tangible programming language (A. F. Blackwell, 2003); earlier I used this framework to interrogate programming with the LittleBits SynthKit (Noble, 2014).

The Cognitive Dimensions framework is a collection various “dimensions” that can be used qualitatively to evaluate a design. The framework is flexible in that there is no canonical set of dimensions, rather existing dimensions can be adopted and new dimensions proposed to suit the task at hand. Following A. F. Blackwell (2003), in this section I evaluate livepatching under twelve commonly-used dimensions, writing the name of the dimension in **boldface** and the key words of the analysis in *italics*.

Medium The medium of expression is primarily the patch cables that link modules; the action of livepatching is ultimately the manipulation of patch cables. The medium of expression also involves the configurations, programs, or knob settings of individual modules. Module parameter knobs are also manipulated in livepatching — more time may be spent tweaking knobs to adjust fine details of a patch rather than the higher-level (and more visible) module patching itself.

Inasmuch as livepatchers (like other Eurorack synthesists) typically select and combine modules to configure their own unique modular synthesizer, this is also part of the medium of expression: the choice of modules, and their layout within rack cases. Unlike patching and knob twiddling, actually building modular synthesizers is not typically part of a livepatched modular performance.

Unlike some other tangible interfaces, patching locations are necessarily *constrained* to the various inputs and outputs on each module, and knob settings are likewise constrained to the options offered by each module. Both livepatching and twiddling expression is *transient*, as cables can be patched or modules twiddled at any time. The transience of the patching is important: as the name implies, it is the

livepatching, rather than the knob twiddling, that distinguishes a livepatched modular performance from a more typical “prepatched” modular performance — where modules parameters will be adjusted, and sounds triggered e.g. using sequences or other gestural controllers, but where the patching is primarily fixed before performance.

Both twiddling and patching are based on the *absolute position* of modules within a rack, and of inputs and outputs within a module.

Activities The key livepatching activity is *incremental construction* and then *modification* of patches, by connecting modules with patch cables, and also *modification* of module parameters by tweaking the parameter knobs. This is essentially *exploratory design* — how exploratory depends on both the courage and ignorance of the synthesist: courage to try something when they’re not sure of the result, and ignorance to increase the likelihood of not being sure of the result.

While modular synthesists may use *transcription* to record patches and module settings, I have not transcribed settings as part of Selective Yellow performances.

Visibility To livepatch a modular synthesizer, the physical synthesizer and patch cables must be to hand: in principle, then, the patch is always *visible*. Unfortunately this principle is not realised in practice: where “spaghetti code” is a metaphor, “spaghetti patching” (or “rat’s nests”) are the rule rather than the exception (see figure 1). While a judicious use of different colours of patch cables and careful module placement within a rack can help mitigate this, it is in practice difficult to “read” a complex modular patch without tracing each individual patch cable — under stage lighting, tracing may mean physically following the path of a cable by touch rather than sight (a novel sense of tangible interface).

Most modules are designed so that the settings of their knobs and switches are visible, at least given sufficient lighting of the performance space (either ambient light, or e.g. small LEDs illuminating the modular). Reasonable visibility of the modular itself is required to successfully patch outputs to inputs. Some modules backlight input and output sockets, mostly to look cool, although this does assist livepatching on stage.

Modules may have other visual outputs — from individual LEDs and 7-segment displays right up to embedded OLED screens (see Figure 3) — to make at least some of their internal state visible. Even individual LEDs can be surprisingly useful: particularly for low frequency oscillators (LFOs) which are primarily used to control other modules. Often the amount of LFO modulation taken into a module will be governed by a separate knob: being able to see where the LFO is in its cycle (maximum, minimum, or somewhere in between) can help setting the right levels to create a particular audio effect.



Figure 3
Oscilloscope.

Diffuseness Any given modular will have a fixed physical size. Most modular cases are designed to fit into standard 19" wide racks, each module occupying 3U of rack space. Individual modules are *moderately sized* ranging from about 1cm wide to as much as 30cm or more. Thus modular patches are relatively *compact* because they have to fit within the available hardware. Even though modulars are larger than most laptops, they are physically smaller than e.g. a pair of large LCD panels used by a professional programmer — and of course, while a program in most languages can be much larger than the available screen real estate, that is not possible with actual hardware. Conversely, the entirety of a modular patch is immediately accessible to the synthesist, whereas editing most programs requires first mapping some parts of the program into the available screen space.

The density of interaction items (input and output sockets, knobs, LEDs etc) varies somewhat across modules, and is limited by the physical size of 2.5mm jacks, LEDs, and knobs. The highest practical density is around one interaction item per couple of square centimetres, with some larger modules being significantly more diffuse — in Figure 1, the left-most “MIDI” module (19 controls in 40cm^2) is significantly more dense than the central “Edges” module (26 controls in 130cm^2). Individual interaction items also vary in size from jack sockets and thin knobs e.g. up to MIDI sockets, LED screens, or knobs that are bigger than usual — either to offer finer control than a smaller knob, for aesthetic reasons, or because twiddling a big knob is somehow more fun than twiddling a smaller one (see Figure 4).

Viscosity Patch cables can be unpatched and repatched between modules, resulting in relatively *low viscosity*. Adjusting module parameters by adjusting knobs is pretty much as direct as possible — the limit being less in actually making changes (viscosity) than in determining which change to make (visibility, above).

Many kinds of changes can involve more than one patch cable, i.e. more than two modules: these kinds of changes encounter rather higher viscosity. Consider an LFO patched to one aspect of a sound (filter cutoff, say) that the performer would like to connect to another aspect of the same sound (amplitude) so that they will be modulated together. This requires the cable linking the LFO and filter to be unpatched, that cable to be repatched into a “multiple” module (which takes a single input and connects it to multiple outputs), then separate cables run from the multiple outputs to the filter and VCA modules. Another common case is inserting one module in the middle of an existing signal path: a drum module could be connected directly to a mixer channel on the way to an output, but the performer would like to process the drum through a low-pass filter modulated by an LFO. This single logical interaction requires three distinct actions (unplugging the output, plugging that into the filter, and then plugging the filter output back into the mixer channel), four actions if fetching another patch cable is counted! Here is one case where tangibility of interaction loosens out to the greater flexibility e.g. offered by virtual modulars such as the Nord (Clavia DMI AB, 1999) which supports multiple connections from outputs and interlinking modules as single interactions.



Figure 4
Verbtronic Module.

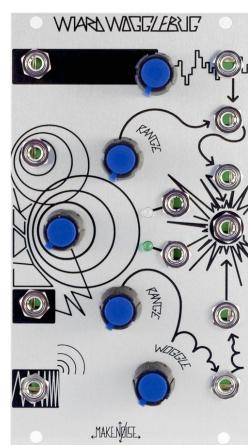


Figure 5
Wogglebug Module.

Secondary Notation Eurorack modules come from a wide range of manufacturers other than Doepfer, so the physical and graphical design varies immensely. Even Doepfer, who began with a very utilitarian grey-on-grey design, have more recently branched out into “special editions” and “vintage series” with different panel and knob colours. The design of the modules themselves is decoupled from their function and serves as a secondary notation to distinguish modules from one another. Depending on their design, modules may be *labelled* with their name, and the functions of the various inputs, outputs, knobs, or other interaction devices — either explicitly with text or simple graphical icons, or implicitly by incorporating controls and sockets as integral parts of a more complex graphical design (see Figure 5). Knob size also serves as secondary notation, making it easier to distinguish between otherwise similar controls. Eurorack modules often follow implicit secondary notation conventions. A very common arrangement is that a parameter of a module is governed by three controls: a knob to set the base value of that parameter (b), a CV input which modulates the value of the parameter (c), and a control knob that attenuates the modulation (m) — the overall value being $b + mc$. The convention here is that the base knob (b) will be larger than the modulation attenuator (m); that ideally all three controls will be placed close to each other (gestalt), or at least the modulation CV input and

attenuator will be placed close together (see the large “OUTPUT MIX” knob, and the smaller “MIX CV IN” knob and socket on the Verbtronic in Figure 4). Where that is not done, graphical elements will link the three controls together (usually from the CV socket *c* to the attenuator *a* then to the base control *b*). Graphically, the base control is a metonymy for the underlying function it performs.

Finally, it is in theory possible to use qualia of the patch cables themselves as secondary notation — different brands of patch cables are different colours, with different finishes (plastic vs fabric). In practice I make little use of this — perhaps because the main supply of patch cables I have use colour coding for length, but more likely because while carefully selecting a cable may be possible in a studio setting, in a livepatching live performance it is difficult to give attention to such details.

The one exception is for initial cabling from the external Beatstep Pro sequencer and control surface: here the patch outputs are on the back of the device, that is, facing away from the performer. I therefore take care to pre-patch cables into the outputs before a performance starts and choose different cable designs for different functions (melodic sequencers vs drum triggers). Even so, more than once I have ended up peering into the back panel of the Beatstep trying to understand why a particular CV signal isn’t behaving the way I expected².

Hidden Dependencies Because every intermodule connection is *explicit*, embodied by a patch cable, there are no structural (or syntactic) hidden dependencies between modules. This true at least in the modular configuration I most commonly use: some particular sets of modules (often the same brand) may support extra connection busses that are not part of the Eurorack specification — cables connecting the rear sides of modules, resulting in dependencies not visible on their faceplates.

Some modules have hidden dependences within their design: so called “normalised” connections that take a signal from an input or output socket in a module into a second input when there is no patch cable in the socket for the second input. For example, the “Edges” module (Figure 2) contains four separate oscillators, each with a separate Gate and Pitch (V/Oct) input socket. The gate and pitch inputs are normalised, so that a single pitch and gate input can control all four oscillators in parallel: additional patching can then take control of each oscillator individually.

The most critical hidden dependencies in modular patching are *semantic*, rather than syntactic. While patch cables establish explicit connections between modules, the interactions between those modules can be much more subtle than a simple connection might imply. A common experience patching modulars (and indeed with much synthesizer programming) is that the patch produces no sound. This can be for as simple a reason as no trigger patched into a gate input, so that oscillators are never switched on, or rather more complex, e.g. if a lowpass filter’s cutoff frequency is below the audio frequency of the oscillator patched into its input, then all that audio will be filtered out. These semantic dependencies are not just hidden, but also silent.

Role Expressiveness Livepatching can be conceived within two distinct roles. First, concretely (or extensionally, or with the performer’s perspective centred), livepatching is primarily what its name suggests: live patching of synthesizers. The domain model is modular synthesis itself: actions within that domain model include “adding in another oscillator” or “adjusting a filter cutoff parameter” or “modulating oscillator pitch by a random stream”. In this role, the *notation models the domain directly* — performers directly interact with the modular synthesizer qua modular synthesizer.

Second, abstractly (or intensionally, or centering the listener) livepatching is manipulating *sound objects* rather than synthesizer modules. The domain is sound, rather than hardware, and activities include “make the sound more plaintive” or “emphasize the bassline” or “descend into a howling vortex of

²There’s an example at about 3min of the video of the *Quiet Noise: Winter* performance, where the pitch of the sequenced oscillator bank didn’t change: it turned out the patch cable going into the oscillator pitch input was plugged into the wrong Beatstep output).

feedback”. In general this role is *not supported* — rather performers must reconceptualise activities in this domain back into the modular domain. Sometimes this mapping can be direct — “emphasize the bassline” may be as simple as turning up the bass in the output mixer; while other times the mapping can be very indirect — “make the sound more plaintive” could require many different manipulations of module parameters, and unpatching and repatching of many modules.

In Selective Yellow, I have found myself *switching* between these roles. Part of this is due to the complexity of constructing the mapping between sound object and synthesizer patching required for the second role; part of this is often choosing to focus on the first role. As with other livecoding practices, demonstrating the underlying construction, emphasizing the “modular-ness” of the sound production, is a key part of the performance: making expansive gestures while patching in a new making module aims to draw attention to the predominance of the first role.

Premature Commitment Repatching a modular synthesizer is essentially the same as patching from scratch, and the generally low viscosity ensures that performers are *not prematurely committed* to particular structures or module settings. Patches can be built (or rebuilt) in any order — subject again to the viscosity issues discussed above: inserting a third module into a signal flow between two modules requires first disconnecting the two modules and then repatching. The physical embodiment of patch cables, and the low cost of repatching, support a *fluid* experience.

The fixed hardware resources of a modular may seem to give rise to premature commitment: if you are using a module for one thing (say an LFO modulating oscillator pitch) you cannot use that same module for something else (say modulating a delay line’s feedback). My typical Selective Yellow modular configuration has eight LFOs: once they are all used, there aren’t any more. This is a problem of *commitment* rather than *prematurity* — an LFO can be unpatched from its current task, and then repatched to perform some other function: this can be one in any order at any time, with only local changes.

Progressive Evaluation Modular patching is by its nature incremental. Sound will be produced so long some audio-rate signal eventually flows to an output module connected to a suitable PA system. Changes to module parameters are typically *evaluated immediately* and become immediately audible — again, provided a suitable signal path from that module to the output.

Accurately patching modules does take time however, so there can be the appearance of a shorter or longer *delay in evaluation*, depending on the performer’s virtuosity in the physical actions of patching. In the *Quiet Noise Winter* performance, for example, the drums are introduced gradually, as individual percussion modules are patched in. Explicit delay modules also induce *delay* (or repetition) in evaluation — presumably a delay intended by the performer (see Figure 6).

Provisionality Cables can be patched and parameter knobs adjusted without being connected to an output, *supporting provisional arrangements* but of course without audible feedback (no progressive evaluation). As mentioned above, for practical reasons I tend to pre-patch connections from sequencer control surfaces simply because making such connections quickly and correctly in the middle of a performance is impractical.

As with many livecoding disciplines I do not use a cue mix to monitor provisional configurations. While separate headphone output modules, or integrated mixer modules with cue busses are available, I choose to work so that “the performer hears only what the audience hears” (Nilson, 2007). If I need to test a particular audio signal, I typically patch that signal into a mixer channel going to the main outputs: the audience can thus hear additional oscillators being brought into the mix, being



Figure 6
Chronoblob.

tuned manually to match the existing sound material, their timbre being adjusted. This way of working is much more feasible for performances in the noise subgenre rather than algorave bangers: in a noise performance an illtuned oscillator can be a welcome feature of the performance, whereas it will at best be distracting (and at worst incompetent) on the dancefloor.



Figure 7 Mult. Modular livepatching does support some abstraction techniques, although they are obviously more tangible (“concrete abstractions”) relying on the use of standard modules. Two kinds of modules are useful here. First “multiples” (aka “mults” or splitters, a single input socket connected to several output sockets, see Figure 7) can allow a single control voltage (or audio source) to be used in several difference places in the synthesizer. A melodic pattern from a sequencer could thus control several different oscillators playing (somewhat) in tune, promoting the sequence into an abstraction in the sense that altering the sequence can alter the performance of a number of different modules. Second, submixers can act in a complementary way, combining a number of audio sources (say all the drum modules) or even a collection of control voltages into a single output. That output can then be processed further “downstream” — perhaps running a drum submix into a single channel of the primary output mixer; or into a flanger or a filter. The downstream controls or processes will now act over the combination of all the audio signals: the knob controlling the main mixer channel that is taking the drum submix now offers a control over all the drums as a single abstractions.

Finally, Eurorack systems (and thus livepatching) can incorporate higher-level abstractions, at the level of rack configurations (module choice) rather than individual patches. Traditional analogue modules provide one function (a single oscillator, filter, or envelope, built from discrete components) however many recent Eurorack modules combine more than one function. A drum module could combine an oscillator, filter, and envelope, yet offer far fewer control parameters, knobs, and input or output sockets than a similar circuit built from individual modules (Figure 8). The reduction in degrees of freedom (“expressibility”) increases ease of use (“musicality”): it is much easier to produce a drum sound from a drum module than it is to patch and configure a whole collection of other modules to produce the same sound, not to mention cheaper. The smaller size of contemporary electronic componentry means that integrated modules can be physically smaller collections of more basic modules, which is important when the synthesizers will be carried into performance venues, rather than being bolted to walls of specialised electronic music studios.

Many contemporary modules digitise incoming audio and control voltages, feed them into a realtime audio algorithm, and then render the output back to analogue signals — evolving towards embedded musical computers rather than single function analogue circuits (Scott, 2016). Interface modules can be used to route audio signals generated within the rack to external signal processors or guitar pedals and then route the resulting audio back into the modular (the hardware equivalent of a foreign function interface). Modules like the AppiOsc (Lawson, Smith, & Appio, 2016) are the logical end-point of this evolution, bi-directionally integrating a modular with a textual live coding environment.



Figure 8 Drum.

4. Discussion and Related Work

In “Live Coding For Free”, McLean (2008) describes how “We can think of coding live in the sense of working on electricity live, re-routing the flows of control around a program with the real danger that a faulty loop will get activated, causing the program to crash in sparks of logic”. Eurorack livepatching works with live electricity — control voltages and audio signals — and loops or misconnections can cause electrical sparks and shorts (although given the low voltages involved, without any serious consequences). McLean also gives three criteria for livecoding. First, *Rules must be explicit* — “written down and modified”. In modular livepatching rules are not written, rather they are embodied in arrangements of patch cables and the settings of sequencers or contents of delay lines: modifying those rules is precisely the point of livepatching. Second, *Higher order functions must be defined and manipulated*: one control signal modulating signal is a higher-order composition of those functions. Third, *An audience is not required*. While I would agree with this philosophically, I have found there is a large difference between rehearsing in private, and performing in public: especially performing to a paying audience (most of who are waiting for the death metal band on next) rather than the privilege of presenting an academic demonstration or educational workshop.

Hutchins (2015) has also investigated livepatching in the wider context of live programming — this paper contributes an analysis based on cognitive dimensions to that investigation. The cognitive dimensions analysis here is similar to an earlier study of livepatching the littleBits SynthKit (Noble, 2014). The key difference is that the Synth Kit is “patched” by physically attaching modules to each other: cabling modules at fixed positions in a rack has quite a different feel. Where manipulating littleBits seemed more object-oriented, my experience of modular patching is more functional, threading streams of values (time-varying control voltages) into functions that produce further value streams. This is in spite of the fact that Eurorack modules often have more internal state than Synth Kit components.

The tension between real and virtual is explicit in comparison with tangible live music programming language systems such as the reacTable (Jordà, Geiger, Alonso, & Kaltenbrunner, 2007), which uses physical objects on a multi-touch table interface to produce music. Animations in the touch table make the physical objects appear live, giving feedback along with the generated sound. In contrast, modular patches are in fact live, tangible, and generate sound directly. Moodler (Piponi, 2015) goes one step further and provides a “Mock Modular” with patch sockets and knobs mounted on a whiteboard so “modules” can be sketched, in order to provide tangible control to a virtual modular in Haskell. Mosaic (Mazza & de Pisón, 2019) moves in the other direction, providing a digital “virtual modular” designed to support livepatching on a laptop computer.

There are still many dimensions of modular livepatching (or indeed modular synthesizer performance) that we have not yet explored. Selective Yellow is a duo, but generally each performer acts independently (other than listening to what the other is doing). We hope to explore the possibilities of multiple performers using a single modular synthesizer, or equivalently, several modular and semi-modular synthesizers being patched together. This seems similar to multiple textual livecoders using computer networks to exchange code fragments during performances, although if anything with more direct interaction. Patching modulars together means one performer can directly affect the sound produced by another, as in Cage’s *Imaginary Landscape No.4* where one performer controls a radio’s tuning, and a second independently controls volume and tone, and where the result is silence as often as sound.

5. Conclusion

In this paper I have reflected on my performance practice livepatching modular synthesizers, through the lens of the cognitive dimensions framework. Treating modular synthesizer as an embodied domain specific programming language has helped identify points that make this practice unique, or at least, positioned at an intersection of two better known practices: livecoding and instrumental improvisation. The tangibility of the modular is the key linkage point, allowing rapid movement between indirect, programmatic patching and configuring (tweaking); and occasional excursions into more direct soloing. The complexity of modular synthesizers, the potential interactions, feedback loops, and higher-order

constructions complements the tangibility: LFOs or random function generators can modulate volume or timbre or any other parameter (including, of course, other modulation paths) and the resulting patches' complexity is limited only by the available hardware. In a Selective Yellow improvisation, this often results in silence (as Cage would have it) but equally often at least some kind of noise.

6. References

- Auricchio, N., & Borg, P. (2016). New modular synthesizers and performance practice. In *SAE Melbourne Symposium*.
- Bjørn, K., & Meyer, C. (2018). *Patch and tweak*. BJOOKS.
- Blackwell, A., McLean, A., Noble, J., & Rohrhuber, J. (2014). Collaboration and learning through live coding (Dagstuhl Seminar 13382). *Dagstuhl Reports*, 3(9), 130–168.
- Blackwell, A. F. (2003). Cognitive dimensions of tangible programming languages. In *First Joint Conference of EASE and PPIG* (pp. 391–405).
- Blackwell, A. F., & Collins, N. (2005). The programming language as a musical instrument. In *Psychology of Programming Interest Group (PPIG)* (pp. 120–130).
- Green, T. (1989). Cognitive dimensions of notations. In A. Sutcliffe & L. Macaulay (Eds.), *People and Computers V* (pp. 443–460). Cambridge.
- Green, T., & Petre, M. (1996). Usability analysis of visual programming environments: A ‘Cognitive Dimensions’ framework. *Journal of Visual Languages and Computing*, 7, 131–174.
- Hutchins, C. (2015). Live patch / live code. In *International Conference on Live Coding (ICLC)*.
- Jordà, S., Geiger, G., Alonso, M., & Kaltenbrunner, M. (2007). The reacTable: exploring the synergy between live music performance and tabletop tangible interfaces. In *Tangible and Embedded Interaction (TEI)*.
- Lawson, S., Smith, R. R., & Appio, F. (2016). Closing the circuit: Live coding the modular synthesizer. In *International Conference on Live Coding (ICLC)*.
- Magnusson, T. (2014). Herding cats: Observing live coding in the wild. *Computer Music Journal*, 38(1), 8–16.
- Mazza, E., & de Pisón, M. J. M. (2019). Mosaic, an openFrameworks based visual patching creative-coding platform. In *International Conference on Live Coding (ICLC)*.
- McKinnon, D. (2011). Centripetal, centrifugal: electroacoustic music. In G. Keam & T. Mitchell (Eds.), *HOME, LAND and SEA: Situating music in Aotearoa New Zealand* (p. 234-244). Pearson.
- McLean, A. (2008). Live coding for free. In A. Mansoux & M. de Valk (Eds.), *FLOSS + Art*. GOTO10.
- McLean, A., Rohrhuber, J., & Collins, N. (2014). Special issue on live coding. *Computer Music Journal*, 38(1).
- Nilson, C. (2007). Live coding practice. In *New Interfaces for Musical Expression (NIME)*.
- Noble, J. (2014). Livecoding the SynthKit: Little Bits as an embodied programming language. In *Software Visualization (VISSOFT)*.
- Nord Modular Manual (V3.0 ed.) [Computer software manual]. (1999). Sweden.
- Paradiso, J. A. (2017). The modular explosion - déjà vu or something new? In *Voltage Connect*.
- Piponi, D. (2015). Moodler: A digital modular synthesiser with an analogue user interface. In *Functional Art, Music, Modelling and Design (FARM)*.
- Rossmay, B., & Wiethoff, A. (2019). The modular backward evolution – why to use outdated technologies. In *New Interfaces for Musical Expression (NIME)*.
- Russell, B. (Ed.). (2012). *Erewhon calling: Experimental Sound in New Zealand*. The Audio Foundation and CMR.
- Scott, R. (2016). Back to the future: On misunderstanding modular synthesizers. *eContact!*(17.4).
- Wilson, C., & Noble, J. (2014). *Selective Yellow*. <https://selectiveyellow.bandcamp.com>.

Acknowledgments

Thanks to Chris Wilson, the better half of Selective Yellow, without whom which.

Mapping the Landscape of Literate Computing

Bjarke Vognstrup Fog Department of Digital Design and Information Studies Aarhus University bfog@cc.au.dk	Clemens Nylandsted Klokmose Department of Digital Design and Information Studies Aarhus University clemens@cavi.au.dk
--	--

Abstract

Literate computing is a computing paradigm that interweaves executable code with more conventional media such as prose, images, and video. It has recently seen uptake particularly in the data science community with tools such as Jupyter Notebook, which is an open source system inspired by Mathematica notebooks. These Mathematica-inspired tools are often referred to as computational notebooks. We, however, argue that computational notebooks are just a special case of literate computing tools and that there is an uncharted design space for computing tools that dissolves the traditional distinction between programming and using computers, but also between using and developing software tools.

We examine various programming environments; some new, some old, some fully developed, some research prototypes. By looking at how we got here—the history—and the challenges identified in the research so far, we analyze the environments through a range of themes, such as purpose, user community, system metaphor, malleability, etc. We conclude with a discussion of design considerations for future literate computing environments.

1. Introduction

Literate computing is human-computer interaction where programming and computation is interwoven with manipulation of text, images, video, and other digital media. The output of literate computing is computational media that in the simplest case allows the reader to rerun computations but in more sophisticated cases provide rich interaction capabilities. The output may be in a narrative form (Pérez & Granger, 2015), take the form of a tool or application, or something in between. A *literate computing environment* is software that is designed to enable literate computing, such as a computational notebook.

The term literate computing is derived from Knuth's *literate programming* (Knuth, 1992) but where Knuth's goal is to make a tool to document code for “for system programmers, not for high school students or for hobbyists”, literate computing environments mix editing and execution of code to create interactive media and computational narratives, and is particularly designed for non-system programmers. One salient attribute of a literate computing environment is that code coexists in the same space as its output and the objects and data it operates on. This attribute is represented visually, for instance in computational notebooks when code cells are adjacent to their output cells in the same document, but also technically when code is addressable as data.

This computing paradigm has in recent years gained significant popularity, most visible in the uptake of computational notebooks in data science. In 2017, according to Rule et al. (2018), GitHub hosted around 1.2 million publicly available computational notebooks, while Kery et al. (2018) established that by 2015, there were already more than two million users of the popular computational notebook, Jupyter. One can only assume that these numbers have grown since then. That makes literate computing a phenomenon that requires scholarly scrutiny: Where does it have its origins? What are its different forms? What are its strengths and weaknesses? What is the scope of its design space? What is its scope of its (potential) user base?

Literate computing has its legacy in the work by diSessa (diSessa, 2001; diSessa & Abelson, 1986), Papert (Papert, 1993), and Kay (A. Kay, n.d.; A. Kay & Goldberg, 1977), among others, that see computational media as not just an extension of conventional media, but rather as a different type of epistemologically interesting media: “what we have to become in order to use it fluently” (A. Kay,

2013).

Literate computing was coined to describe the particular kind of human-computer interaction present in computational notebooks (Pérez & Granger, 2015). However, we argue that computational notebooks are just one particular expression of the literate computing genre, and that it is worth exploring the broader potentials of the genre. In this paper we will give an overview of the genealogy of literate computing, present the current state of the art of technologies and research and discuss future directions both in terms of the design and study of literate computing environments.

2. Literate computing environments

In this section we will present a range of software environments that enable literate computing or exhibit characteristics related to literate computing. They are chosen for their popularity, novelty, historical importance or because they incorporate specific design choices. We will divide them into three separate but overlapping groups: *Historical* environments that have exemplary significance but generally no longer see widespread use; *in-use* environments that currently have a broad real-world uptake; experimental *research* prototypes or proof-of-concepts that explore certain aspects of literate computing and push the state-of-the-art forward.

2.1. Historical

2.1.1. Boxer

Boxer, created by Andrea diSessa and Harold Abelson in 1986, was presented as a *reconstructible computational medium*. Starting from the premise that programming could (and even should) be a commonplace skill like reading and writing, Boxer provides an environment for non-professionals to program: “Not only would professionals be able to construct grand images, but others would be able to reconstruct personalized versions of these same images” (diSessa & Abelson, 1986).

Boxer is a literate computing environment, and an example of how literate computing does not only equate computational notebooks. Whereas the typical notebook is structured linearly in one dimension, Boxer allows for creating a two dimensional organization of content, resembling whiteboards more than notebooks. Its user interface is simplistic: Everything is represented as (graphical) boxes that might contain code, prose, data, or other types of media, and the system is based on *naïve realism*—only what you see in the canvas is what is there. Like Seymour Papert with LOGO before him, diSessa has since done significant research on the use of Boxer in educational settings, investigating how schoolchildren might appropriate the system for learning and exploration (diSessa, 2001).

2.1.2. HyperCard

HyperCard is not a literate computing environment *per se*, as code is not a first-class citizen but hidden as scripts behind the scenes. HyperCard is, however, a computational media authoring environment that seeks to close the gap between the developer and the user. In this respect, it is the intention behind HyperCard that is of interest. HyperCard was inspired by hypertext as an organizing principle, and in turn inspired early web browsers.

As a programming environment, HyperCard was particular due to the fact that all data is essentially string-based (*A Eulogy for HyperCard*, 2004). In spite of this, several commercial games, such as *Cosmic Osmo* and *Myst*, were developed through the platform. The most interesting aspect of HyperCard is the ease with which it could be used to construct software, even for non-programmers. By providing an interface builder and a scripting language for creating interactivity, HyperTalk, HyperCard brought programming-like abilities to the average household.

2.1.3. Smalltalk

Smalltalk (A. C. Kay, 1993) deserves to be mentioned here, if only for its complete commitment to the *liveness* of software development. Following the object-oriented programming paradigm to its fullest, everything in Smalltalk is simultaneously represented as a conceptual object and a visible GUI element. Further, Smalltalk embodied a concept of liveness that is perhaps more likened to reactive programming, enabling a user to edit the running system from within said system. No compilations or restarts

required. These twin concepts of liveness and malleability are reappearing in several literate computing environments, as we will see later on.

2.2. In use

2.2.1. Mathematica

Mathematica was released in 1988 and is built around the Wolfram computational language. The creator, Stephen Wolfram, argues that Wolfram is not only a programming language as it is “not just a language for telling computers what to do. It’s a language that both computers and humans can use to represent computational ways of thinking about things” (*Stephen Wolfram Blog*, 2019).

Since its inception, Mathematica has relied on a notebook format in the form of *computational essays* containing code and other media (Hayes, 1990), and its cell-based format has set the standard for the structure of other computational notebooks. Notably, Mathematica is the only non-historical platform on this list that is not free to use.

2.2.2. Jupyter Notebook

Jupyter Notebook is the most well-known computational notebook today. Previously called IPython, this environment has been around since 2007 (Pérez & Granger, 2007) and was originally based on the popular, open source language Python. While later made available as a web service through Google Colab¹, the environment is designed to be installed on a local machine and used by a single user through a web browser using an HTML based user interface.

Jupyter Notebook employs a conventional file-based structure, imitating the underlying structure of the operating system. However, the files themselves are akin to Knuth’s WEB-files (Knuth, 1992) as they contain prose, code, images, etc. all at once. The content is structured into separate cells that can be run or viewed individually. Importantly, Jupyter is by far the most popular interactive notebook for data scientists (Perkel, 2018). Today, Jupyter supports over 100 languages². Jupyter has an active community, and a myriad of extensions exist to tweak the user interface or to, e.g., create slide presentations from a notebook.

2.2.3. Observable

Observable is presented by its creator as “a better way to code” (Bostock, 2017). Like Jupyter Notebook, Observable is a cell-based interactive notebook. However, a key difference is that Observable is web-based and employs a *reactive* programming paradigm in a JavaScript-like language. This allows for instant feedback and updates in the entire document. A notable feature of Observable is how the platform does not distinguish between editing and use—when viewing a notebook, the user can directly edit said notebook and fork it as their own.

Observable takes its departure in data-driven development through D3.js, a data analysis framework. The major difference from the other environments is Observable’s graphical representations and inherent focus on data visualization (Bostock, 2018). Through its reactive nature and the ease of which sophisticated interactive visualizations can be embedded, it enables the creation of narratives resembling Victor’s exploratory explanations (Victor, 2011).

2.2.4. Notion

Although not a programming environment, Notion brings into play some of the design considerations that characterize literate computing. Sitting somewhere among Trello and Slack, Notion serves as an integrated platform for documents, data and media.

It is web-based, but provides desktop shells for popular operating systems, including mobile, that makes it at least feel like a stand-alone application. Particularly interesting in the current context is its script-like capabilities that are closely aligned with Raskin’s notion of *commands* (Raskin, 2000, p. 109). These commands, such as @Today, execute automatically upon being entered—in this case, the command inserts today’s date as a dynamic element. The next day, the string will instead read @Yesterday.

¹colab.research.google.com

²<https://github.com/jupyter/jupyter/wiki/Jupyter-kernels>

While not technologically ground-breaking, the example serves to illustrate the potential for Notion to enhance a text-based medium with computational capabilities.

2.3. Experimental or research prototypes

2.3.1. iodide

A web-based data science tool, iodide³ is closely aligned with Knuth’s vision of a literate programming environment. The environment presents three different windows: the editor, a preview, and console/workspace. The editor looks like any other text-based code editor with consecutive lines. However, by writing `%%` followed by a certain command, one can insert a divider to change between a range of modes from that line forward. This is a particularly interesting feature of iodide: By changing modes, the user can write in Python, JavaScript, CSS, Markdown, HTML, or call `fetch` to import external data. The editor supports syntax highlighting for all modes. iodide is fully reactive; any change made in the editor is instantaneously reflected in the preview. Lastly, an iodide document is located at a specific URL that can be shared like any other address.

2.3.2. Live notebook

Live notebook⁴ is an evolution of the principles from Jupyter Notebook. Live notebook employs a reactive programming environment—that is, cells that are linked together update automatically across the environment. Live notebook also supports native real-time collaboration. And finally, Live notebook provides a WYSIWYG editor and thus eliminates the need to know Markdown, HTML or a similar markup language.

2.3.3. Codestrates

Codestrates (Rädle, Nouwens, Antonsen, Eagan, & Klokmose, 2017) comes from an academic effort to provide new tools for computational thinking. Based on Webstrates (Klokmose, Eagan, Baader, Mackay, & Beaudouin-Lafon, 2015), a dynamic shareable medium, Codestrates operates as a purely web-based environment. Codestrates adopts a structure similar to word processors. Content (e.g., code, prose, data) is structured in sections that can be collapsed and unfolded. The programming language in Codestrates is JavaScript simply because it is the programming language of the web. One of Codestrates’ trademarks is that the environment itself is extensible in the spirit of diSessa and Kay. If a function is missing, you can program that function yourself, and every notebook (called a codestrate) is self-contained similar to a Smalltalk image. Code can be shared between codestrates as packages (Borowski, Rädle, & Klokmose, 2018).

Vistrates (Badam, Mathisen, Radle, Klokmose, & Elmqvist, 2019) is an extension to Codestrates that allows users to create a reactive data processing and visualization pipeline. In Vistrates, a non-programmer might never write code themselves and still be able to construct pipelines of data processing and visualizations and aggregate these on an interactive canvas for presentation. Meanwhile, a programmer can unfold the code, edit and create components. In Vistrates, multiple users can simultaneously interact with and reprogram a shared reactive visualisation pipeline.

2.3.4. Eve

Eve is the product of Chris Granger, former lead developer of Visual Studio, who tried to create a “human-first programming platform”⁵. Speaking of Visual Studio, Granger remarked how “it missed the fundamental problems that people faced. And the biggest one that I took away from it was that basically people are playing computer inside their head.” (Somers, 2017).

Eve was an attempt to rectify this, although sadly short-lived. Its files embodied a modular approach through which a document was assembled from smaller blocks—what Granger calls “programming by composition”. The language of Eve worked through pattern matching, where each block of code was a named group that was matched in real-time with a database to create relations between blocks. This allowed for a novel and reactive programming environment. Eve programs were structured like essays,

³<https://alpha.iodide.io/>

⁴<https://livebook.inkandswitch.com/>

⁵<http://witheve.com/>

mixing prose and code modules and showing the resulting graphical application in an adjacent pane.

2.3.5. Capstone

Capstone⁶ provides a 2D environment for creative professionals that can be understood as a whiteboard-like canvas with cards. One interesting feature has brought it here: bots. Through a series of experiments in end-user programming, Capstone was augmented with embodied programs (*End-user programming*, 2019). Embodied in the sense that these programs exist on the screen as visual elements. A bot is, then, a daemon-like service that provides computational capabilities, such as automatically reordering the boards on the canvas. Importantly, bots can be created by users themselves to augment their workflow. Further, users can program not only the bots' functionality, but also their visual representation and interactivity.

2.4. Research on literate computing

The research done on literate computing is surprisingly sparse, considering the popularity of computational notebooks in both industry and academia. Broadly, we might distinguish between two kinds of research while bearing in mind that there is a great deal of overlap between them: 1) the use of literate computing environments, and 2) the design of literate computing environments, respectively.

2.4.1. Use

Much research on literate computing environments tend to investigate the use of existing tools. Kery and Myers (Kery & Myers, 2018) have explored how data scientists keep track of notebook versions, while Kery et al. have looked into how scientists curate notebooks into proper narratives (Kery et al., 2018).

Rule et al. investigated how academic data analysts view their own notebooks as “personal, exploratory, and messy”, and how they subsequently share the results of their analyses (Rule, Tabard, & Hollan, 2018). Rule’s PhD dissertation (2018) likewise shows how people use computational notebooks and goes on to present several design solutions to the obstacles found. Owing to the massive popularity of Jupyter Notebook, Shen and Perkel sought to present the particularities and possibilities of the environment to a wider audience through articles in Nature (Shen, 2014; Perkel, 2018).

2.4.2. Design

Research efforts in literate computing often culminates in new systems that explore possibilities, limits, and opportunities. One such system was Pérez and Granger’s IPython (predecessor to Jupyter), presented to their peers in the natural sciences (Pérez & Granger, 2007). More recently, Rädle et al. have developed Codestrates—a shareable, dynamic literate computing environment as part of a wider research effort on developing new tools for computational thinking (Rädle et al., 2017).

Another group of research projects seek to augment existing tools with new capabilities. Rule et al. have investigated how a computational notebook might be structured and collaboratively re-used through cell-folding (Rule, Drosos, Tabard, & Hollan, 2018). Head et al. recently presented extensions to computational notebook environments for “code gathering”, helping data scientists de-clutter and organize their work (Head, Hohman, Barik, Drucker, & DeLine, 2019).

Criticizing the linear narrative structure that is often implied in computational notebooks, Mathisen et al. have presented InsideInsights (Mathisen, Horak, Klokmose, Grønbæk, & Elmquist, 2019) that is structured around multiple non-linear narratives, eschewing the data layer in favor of a narrative layer. Wood et al. likewise favor a more complex storytelling featuring branching narratives and a focus on *literate visualizations* (Wood, Kachkaev, & Dykes, 2019).

3. Themes

It is yet too early to create a comprehensive taxonomy on literate computing environments (comparable to, e.g., Kelleher & Pausch, 2005). However, we now present seven themes that can frame an analysis of these environments and provide directions for future research.

⁶<https://www.inkandswitch.com/capstone-manuscript.html>

3.1. System metaphor

A system metaphor can be understood as “how do I make sense of this thing” by providing a well-known idea, concept, or artifact through which *this thing* can be made meaningful. This is referred to as, among other names, conceptual blending (Fauconnier & Turner, 2003). A conceptual blend arises from the combination of two conceptual inputs, for example the idea of an ordinary desktop mixed with the file system of a computer.

Boxer and HyperCard, two discontinued platforms, utilize a canvas-like metaphor and a stack of cards, respectively. In contrast, most contemporary literate computing environments draw upon the idea of a *notebook*—for some, it’s even part of their names. This is characterized by a certain layout that forces a vertical, linear structure upon the work. Even if the linearity can be discontinued, e.g., by running code cells in a different order, the linear structure still persists, causing confusion: “Did I already run this cell?”. In their study of more than a million Jupyter Notebooks, Rule et al. (2018) point out that almost half of these are non-linear in their execution. This is one of the many challenges inherent in the notebook format. Another is that the notebook metaphor does not support multiple purposes well, such as data exploration and data explanation, at the same time (Mathisen et al., 2019).

Codestrates, Mathematica and EVE are more akin to a Word-like document. For example, Codestrates structures the page in sections and paragraphs rather than cells. They do still, however, implement a vertical, linear structure. There is an inherent clash between the computer’s non-linear execution model and the linear narratives presented in most current literate computing software. Functions, a core concept in most programming languages, violate the expectation of code running from line 1 to line N linearly—something most teachers in programming probably have experienced student confusion about. The difference between a notebook metaphor and a text document metaphor might seem superficial. However, the metaphor clearly signifies intention as we will see next.

3.2. Intention & Purpose

We might at a high level distinguish between multiple goals of programming: programming for computation, programming for interaction and programming for software engineering. Data science falls into the first category; the purpose of programming is not to construct running systems, but rather to use the environment as a glorified calculator and compute, evaluate, and present a given slice of the world. In contrast, programming for interaction is making a certain system come alive. In HyperCard, programming scripts did just that. Programming for software engineering is what some non-programmers would probably imagine a typical programmer doing—writing more or less complex software that can be installed, run, and utilized.

Characteristically, the platforms that embody a notebook metaphor typically have their roots in a very specific form of computing: data science. Data exploration, for instance, can be—and often is—a (sub-)goal for data scientists. Through data exploration, scientists *make sense* of their data, often followed by its counterpart activity, explanation (Rule, Tabard, & Hollan, 2018). That is, the subsequent sharing, presentation, or discussion of the findings from data analysis. A similar notion is found in the concept of *explorable explanations*—however, this concept entails creating finished compositions in which the end-user might explore an interactive paper, for instance. This brings it closer to the concept of computational media in general, which we might also see reflected in Eve and Observable.

3.3. Threshold & ceiling

These labels refers to how easy is it for programmers and non-programmers to get hold of, appropriate, learn, and eventually master a given system. Myers, Hudson, and Pausch (2000) offer the terms *threshold* and *ceiling* to illustrate, respectively, how difficult it is to learn to use a system, and how much can be done with it. They remark that most successful systems often have either low threshold and low ceiling or high threshold and high ceiling. HyperCard was exemplary for its low threshold—you could construct complicated systems without knowing much—or even anything—about programming. For instance, many educators in Melbourne took up HyperCard to create classroom activities and educational software (Lasar, 2019). Not considering the high price its and proprietary computational language,

Mathematica employs a very powerful symbolic manipulation. Once you know the basic syntax, (almost) everything can be expressed and computed.

In contrast, Observable has a much higher threshold. The language is JavaScript-but-not-quite, meaning there is likely a transitional phase for most people. Jupyter (and Python) is often lauded for its ease-of-uptake, but installing the environment still requires particular knowledge and advanced computer skills such as terminal use. Importantly, threshold and accessibility are very subjective measures. Theoretically, any system that is Turing complete can do anything that another can, while people naturally have different thresholds for systems. The question of threshold and ceiling is still valid, though. Even though you could build a 3D-shooter in Excel, does not mean that you ought to.

3.4. Liveness

There are several aspects of liveness, the first of which is related to how the environment behaves. Does it feel static, running code once, doing something, then coming to a stand-still? Or does it instead react to user input in a real-time manner, affording a dynamic, interactive environment. In other words, does the program *live*, in the sense that it has state, provides feedback, and exists over time? Some systems, such as Observable, Live notebook, Eve, and iodide utilize the latter principle to its fullest. Just the act of adding or removing a character triggers immediate reactions throughout the environment. These systems are *responsive*, but not truly *live*, as defined in Tanimoto's four levels of liveness (Tanimoto, 1990).

Liveness is a fickle attribute and might likewise refer to (re)-programmability. According to Basman et al. (2016), software can be considered live when the *divergence* is closed. Divergence is here defined as the opposition between the running software and the mental representation of said software. Hidden states, as for instance in Jupyter, might increase divergence, while a system like Boxer specifically sought to avoid this through, e.g., naïve realism.

3.5. Malleability & extensibility

How malleable and extensible an environment is, is really about its plasticity, both from the inside and outside. A classic example is Excel, allowing users to program their own extensions through macros. Likewise, the seminal text editor emacs allows users to extend the software in real-time by programming new functionality.

Codestrates is the most obvious example of a fully extensible environment in our list—the introduction of packages (Borowski et al., 2018) provides an easy way for developers to write extensions to the system itself through repositories. In contrast, environments such as Jupyter Notebook and Live notebook, while publicly available as source code, requires the user to edit said code from outside the system in a different editor. And some, like Mathematica or Observable, are impossible to change due to their being proprietary.

3.6. User community

We can divide the user community into two parts, the intended versus the actual community. It is not given that these are the same or even overlap. As a product of university research, Codestrates has mainly been taken up by researchers themselves and by their students. However, its flexibility makes it well-suited for many uses, and currently there are ongoing projects adapting Codestrates to citizen science, lab notebooks and high school education. In contrast, Jupyter Notebook was created by natural scientists for their community peers in the sciences. Like Mathematica, this creates a certain community—however, Mathematica's price tag clearly separates the actual user groups.

Two of the more interesting examples are Observable and HyperCard. Observable is explicitly oriented towards visualizations (Bostock, 2018), implying a user community that needs to communicate data analysis to outsiders, e.g., journalists. HyperCard, famously, was taken up by people who did not already know how to program. In doing so, many people ended up as happenstance programmers as part of a user community that was ill-defined from the start (*A Eulogy for HyperCard*, 2004).

3.7. Collaboration

Collaboration relates to the ways that collaboration is allowed, or even possible through an environment. For example, Observable and Jupyter Notebook are to a high degree based on solitary work practices. Recently, however, both tools have acquired after-the-fact support for collaboration either through Google Colab or natively (Ashkenas, 2018). In contrast, Codestrates and Live notebook natively support real-time collaboration.

There are other aspects of collaboration than real-time, equal collaboration. For instance, asynchronous collaboration ideally requires tools for managing and viewing changes that have happened between access. There are also different levels of *equality* in collaboration. Two people might collaborate equally, synchronously or asynchronously, each having the same access and responsibility. In contrast, one can also imagine, for example, a teacher collaborating with students, where the level of participation is unequal—maybe the teacher should only be able to read and comment on students' code. Vistrates is one environment that explicitly explores collaboration between unequal participants, where a programmer can edit the code of a visualization while a user interacts with it on another machine.

This theme links back to the systems' intended purposes—are they made for a single developer to write applications, do they support presenting data analysis findings, and does the collaboration imply an equal stake in the work?

3.8. The future of literate computing

We have presented a view of literate computing environments as it currently presents itself and brought to the fore seven salient themes. These are, of course, not exhaustive. Further, we would like to point at possible futures of these and other systems, as the design space of literate computing environments is far from fully explored.

The most prevalent expression of literate computing right now is the computational notebook. However, as we have presented, it has its drawbacks, such as the implied linear structure and the insistence in supporting *one* computational narrative. We see potential in expanding on the work by, e.g., Wood et al. (2019), which incorporates a more nuanced view of the narrative as a structuring factor. We might also do away with the narrative altogether, and instead scaffold literate computing systems with a spatial metaphor in line with, e.g., Boxer. Here, we would like to once again point to the software execution model. The idea of linearity is a myth in contemporary computing, and so we need to experiment with different forms of program flow. Vistrates, for instance, makes explicit the execution model by enforcing the creation of pipelines to structure the running program code.

Most current literate computing tools support only programming for computation. While it *is* possible, you would be hard pressed to develop sophisticated software architecture in Jupyter Notebook. It would likely prove fruitful to explore the borderlands between different goals of programming. One system does not need to support all goals. For instance, Jupyter Notebook is well-suited for programming for computation, but not for software engineering. In contrast, Codestrates is highly extensible and re-configurable, but this seems to hinder the ease of doing programming for computation.

Newer literate computing environments employ a reactive model in which any changes made are instantly reflected across the system. They can, however, still contain hidden states, obscuring the system's inner workings. We might look towards Eve's *absolute transparency* effort, Boxer's concept of naïve realism, or Bret Victor's essay on *learnable programming*⁷ for inspiration on how to make the systems less opaque. Finally, we suggest that new system metaphors might serve as a starting point in exploring what literate computing could also be. The environments presented in this paper mainly employ notebook or essay metaphors. Just as the desktop metaphor in computing is ubiquitous but not essential, we might look to other real-world phenomena around which to construct new systems.

⁷<http://worrydream.com/\#!/LearnableProgramming>

4. References

- Ashkenas, J. (2018, November). *Fork, Share, Merge*. Retrieved from <https://observablehq.com/@observablehq/fork-share-merge>
- Badam, S. K., Mathisen, A., Radle, R., Klokmose, C. N., & Elmquist, N. (2019, Jan). Vistrates: A component model for ubiquitous analytics. *IEEE Transactions on Visualization and Computer Graphics*, 25(1), 586–596. doi: 10.1109/tvcg.2018.2865144
- Basman, A., Church, L., Klokmose, C. N., & Clark, C. B. D. (2016). Software and how it lives on - embedding live programs in the world around them. In *Proceedings of ppig 2016*.
- Borowski, M., Rädle, R., & Klokmose, C. N. (2018). Codestrate packages: An alternative to "one-size-fits-all" software. In *Extended abstracts of the 2018 chi conference on human factors in computing systems* (pp. LBW103:1–LBW103:6). New York, NY, USA: ACM. doi: 10.1145/3170427.3188563
- Bostock, M. (2017, April). *A Better Way to Code*. Retrieved from <https://medium.com/@mbostock/a-better-way-to-code-2b1d2876a3a0>
- Bostock, M. (2018, March). *Why Observable?* Retrieved from <https://observablehq.com/@observablehq/why-observable>
- diSessa, A. A. (2001). *Changing minds: Computers, learning, and literacy*. Cambridge, Mass.: MIT Press. (OCLC: 464582529)
- diSessa, A. A., & Abelson, H. (1986, September). Boxer: A reconstructible computational medium. *Communications of the ACM*, 29(9), 859-868. doi: 10.1145/6592.6595
- End-user programming*. (2019, Mar). Retrieved from <https://www.inkandswitch.com/end-user-programming.html>
- A eulogy for hypercard*. (2004, Mar). Retrieved from https://due-diligence.typepad.com/blog/2004/03/a_eulogy_for_hy.html
- Fauconnier, G., & Turner, M. (2003). *The way we think: Conceptual blending and the mind's hidden complexities* (1. paperback ed ed.). New York, NY: Basic Books.
- Hayes, B. (1990). Thoughts on mathematica. *Pixel: the magazine of scientific visualization*, 1(1), 28–35.
- Head, A., Hohman, F., Barik, T., Drucker, S. M., & DeLine, R. (2019). Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19* (p. 1-12). Glasgow, Scotland Uk: ACM Press. doi: 10.1145/3290605.3300500
- Kay, A. (n.d.). Afterword: What Is A Dynabook? , 9.
- Kay, A. (2013). The Future of Reading Depends on the Future of Learning Difficult to Learn Things. *VPRI Related Writings*.
- Kay, A., & Goldberg, A. (1977, March). Personal Dynamic Media. *Computer*, 10(3), 31-41. doi: 10.1109/C-M.1977.217672
- Kay, A. C. (1993, March). The early history of Smalltalk. *ACM SIGPLAN Notices*, 28(3), 69-95. doi: 10.1145/155360.155364
- Kelleher, C., & Pausch, R. (2005, June). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2), 83-137. doi: 10.1145/1089733.1089734
- Kery, M. B., & Myers, B. A. (2018, October). Interactions for Untangling Messy History in a Computational Notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (p. 147-155). Lisbon: IEEE. doi: 10.1109/VLHCC.2018.8506576
- Kery, M. B., Radensky, M., Arya, M., John, B. E., & Myers, B. A. (2018). The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18* (p. 1-11). Montreal QC, Canada: ACM Press. doi: 10.1145/3173574.3173748
- Klokmose, C. N., Eagan, J. R., Baader, S., Mackay, W., & Beaudouin-Lafon, M. (2015). *Webstrates: Shareable Dynamic Media*. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology - UIST '15* (p. 280-290). Daegu, Kyungpook, Republic of Korea: ACM

- Press. doi: 10.1145/2807442.2807446
- Knuth, D. E. (1992). *Literate programming* (No. no. 27). Stanford, Calif.: Center for the Study of Language and Information.
- Lasar, M. (2019, May). *30-plus years of hypercard, the missing link to the web*. Retrieved from <https://arstechnica.com/gadgets/2019/05/25-years-of-hypercard-the-missing-link-to-the-web/>
- Mathisen, A., Horak, T., Klokmose, C. N., Grønbæk, K., & Elmquist, N. (2019). Insideinsights: Integrating data-driven reporting in collaborative visual analytics. *Computer Graphics Forum*, 38(3).
- Myers, B., Hudson, S. E., & Pausch, R. (2000, March). Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1), 3–28. doi: 10.1145/344949.344959
- Papert, S. (1993). *Mindstorms: Children, computers, and powerful ideas* (2nd edition ed.). New York, NY: Basic Books. (OCLC: 28504839)
- Pérez, F., & Granger, B. (2007). IPython: A System for Interactive Scientific Computing. *Computing in Science and Engineering*, 9(3), 21-23.
- Pérez, F., & Granger, B. (2015). *Project Jupyter: Computational Narratives as the Engine of Collaborative Data Science*.
- Perkel, J. M. (2018, Oct). Why jupyter is data scientists' computational notebook of choice. *Nature*, 563(7729), 145–146. Retrieved from <http://dx.doi.org/10.1038/d41586-018-07196-1> doi: 10.1038/d41586-018-07196-1
- Rädle, R., Nouwens, M., Antonsen, K., Eagan, J. R., & Klokmose, C. N. (2017). Codestrates: Literate Computing with Webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology - UIST '17* (p. 715-725). Québec City, QC, Canada: ACM Press. doi: 10.1145/3126594.3126642
- Raskin, J. (2000). *The humane interface: New directions for designing interactive systems*. Reading, Mass: Addison Wesley.
- Rule, A. (2018). Design and Use of Computational Notebooks. *UC San Diego Electronic Theses and Dissertations*.
- Rule, A., Drosos, I., Tabard, A., & Hollan, J. D. (2018, November). Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW), 1-12. doi: 10.1145/3274419
- Rule, A., Tabard, A., & Hollan, J. D. (2018). Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18* (p. 1-12). Montreal QC, Canada: ACM Press. doi: 10.1145/3173574.3173606
- Shen, H. (2014, Nov). Interactive notebooks: Sharing the code. *Nature*, 515(7525), 151–152. Retrieved from <http://dx.doi.org/10.1038/515151a> doi: 10.1038/515151a
- Somers, J. (2017, September). The Coming Software Apocalypse. *The Atlantic*.
- Stephen wolfram blog. (2019, May). Retrieved from <https://blog.stephenwolfram.com/2019/05/what-weve-built-is-a-computational-language-and-thats-very-important/>
- Tanimoto, S. L. (1990). Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2), 127 - 139. doi: [https://doi.org/10.1016/S1045-926X\(05\)80012-6](https://doi.org/10.1016/S1045-926X(05)80012-6)
- Victor, B. (2011, Mar). *Explorable explanations*. Retrieved from <http://worrydream.com/ExplorableExplanations/>
- Wood, J., Kachkaev, A., & Dykes, J. (2019, January). Design Exposition with Literate Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 25(1), 759-768. doi: 10.1109/TVCG.2018.2864836