Investigating and Automating the Creative Act of Software Engineering

Simon Colton[1, 2], **Edward J. Powley**[1] and **Michael Cook**[1]
[1]The MetaMakers Institute, Falmouth University, UK
[2]Computational Creativity Group, Goldsmiths, University of London, UK
www.metamakersinstitute.com ccg.doc.gold.ac.uk

Abstract

We take the position that the creative act of computer programming has been under-investigated in Computational Creativity research. It is time for a concerted study of software engineering from the perspective of creative software. Such software should produce code and algorithms as artefacts of interest in their own right, rather than as a means to an end. We briefly survey and critique existing automated programming approaches, propose some novel methods for this, and investigate potential application areas for automated software engineering. Central to our position is the notion that creative software generators should perform in an unsupervised manner in order to problematise the world rather than (or in addition to) solving given problems. This will necessarily utilise some current methodologies and philosophies from Computational Creativity research, and we explore the ways in which these could guide future software synthesis approaches.

Introduction and Motivation

Creative people write software for a number of purposes. Often, coding is a means to an end of achieving some goal, automating some task or solving some problem. In these cases, the value of the written code is secondary to the value of running the code in a particular application. However, in other contexts, the software itself is appreciated as an important creation or discovery over and above any application of it (if there is one). As discussed below, examples of where this is the case include scientific discovery, automated creators, recreational coding, and games (for education or entertainment) which use coding as a game mechanic.

Like a product or process in the arts or sciences, code can take on a life of its own, being studied, modified, used in unforeseen applications and even celebrated culturally. We explore here the position that Computational Creativity research would be well served by thinking of computer programs as important artefacts in their own right, rather than purely as task-completing or problem-solving processes. We therefore advocate studying and automating the creative act of software engineering, similar to studying and automating the creative act of painting, writing, composing, etc.

It is tempting to point out that, as Computational Creativity researchers, we build software to generate artefacts such as paintings, poems, musical compositions, videogames, etc., and hence automating the engineering of such software systems would represent a meta-approach of value to the field. However, it is too ambitious currently to suggest the automatic production of generative systems in all but highly specialised applications. Moreover, this would obscure our point that the code artefacts themselves, rather than outputs from running the code, should be the end goal for implementations which simulate creativity in programming.

Critical to our outlook for the study of automating creativity in programming is the notion that it should be used to **problematise the world** rather than (or in addition to) solving given problems. By *problematise*, we mean that the generated code exposes opportunities either for better understanding the world through problem solving (e.g., the code exposes an unexpected anomaly or hypothesis about a dataset), or application of the code in cultural contexts to change the world (e.g., the code can be used as a mechanic in a videogame). In both cases, it is important to note that the generated code needs to be appreciated as a cultural artefact in its own right, for its aesthetic and knowledge-enhancing qualities and not just for its utility, in order for the opportunities to be fully understood and exploited.

As a hypothetical example, imagine in the early days of computer graphics that an automated code synthesis system had output an image filtering algorithm which performed edge detection, i.e., similar to the invention by Canny (1986). Imagine further that this was a completely unexpected piece of code, i.e., the user had no idea in advance that edge detection was even something that software could do. In this hypothetical context, the discovery of this algorithm would lead to a series of problems going from **unknown unknowns**[1] to known unknowns. In other words, the new discovery does not immediately provide all the answers about this new domain, but it does expose which questions are interesting for further study. Such exposed questions would include: how does this algorithm work; how good is the edge detection and how do we measure this; what are the practical applications of this software; what are the artis-

[1]The term 'unknown unknowns' is originally attributed to psychologists Luft and Ingham in their development of the Johari window technique. However, it was brought to popular attention by US Secretary of Defense Donald Rumsfeld in 2002, with the statement: "We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns – the ones we don't know we don't know."

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n=(count+7)/8;
    switch(count%8){
    case 0: do{ *to = *from++;
    case 7:     *to = *from++;
    case 6:     *to = *from++;
    case 5:     *to = *from++;
    case 4:     *to = *from++;
    case 3:     *to = *from++;
    case 2:     *to = *from++;
    case 1:     *to = *from++;
        }while(--n>0);
    }
}
```

Figure 1: Duff's device (Duff 1988).

tic affordances of this software; what are other approaches to edge detection, etc. Solving some of these problems will lead to better understanding the world, while others will lead to new practical opportunities. Moreover, studying the code as an important cultural artefact in its own right may itself lead to the generalisation and formalisation of the edge detection process found in the generated code in language independent terms. It is not too far fetched to make an analogy here with formative artworks by an artist – while better algorithms may follow, having the original algorithm in the world is important historically and culturally.

Currently, the responsibility for asking these follow-up questions falls to the people who set the code generation system in motion and who study its output (which assumes that the output is comprehensible to humans – by no means a given). It is interesting to imagine an AI system which is capable of appraising generated code in this way.

The edge detection example above considers software from a functional perspective: the interesting part is the operation the code performs. However, the same principle can apply to more abstract code structures, where the the code itself is interesting (within the application domain of software engineering rather than elsewhere). For an historical example, Duff's device (1988) is a clever abuse of the `switch-case` statement in C (see figure 1), which allows a loop to be partially unrolled (often resulting in faster execution) without introducing any restrictions on the number of iterations the loop may execute. As previously, imagine this had been discovered not by a software engineer at Lucasfilm, but by an automated programming system. It similarly exposes new problems: how does it work; is it general; how much efficiency gain does it yield? Solving these may lead to new discoveries by human or by computer, just as Duff's device inspired a particularly elegant implementation of the coroutine idiom in C, as described by Tatham (2000).

In both these examples, an obviously useful piece of software has highlighted many new and interesting problems. While automated programming clearly has problem-solving applications, we believe that applications which expose unknown unknowns are key to modelling and utilising creative behaviours, which in turn is essential to studying the full potential of automated approaches to code generation. Creative behaviours free the approach from difficult constraints, but in turn introduce a number of difficulties in execution, which are discussed below. We believe that creative automated coding approaches will be able to enhance the arts, lead to scientific breakthroughs and drive progress in society. We further believe that advances in automated programming have been held back by the almost-universal application of them within the problem-solving rather than artefact generation paradigm of AI. Bringing creative automated coding into Computational Creativity research would open new avenues of research, provide a step change in the value of artefacts being generated and unearth new application domains. We would expect to see more interesting and sophisticated processes being undertaken by software, advancing our understanding of what it means for software to be creative.

The rest of the paper is organised as follows. In the next section, we look at various ways in which programming has been automated, and provide a critique highlighting the inappropriateness of these methods with respect to creative affordances and cultural celebration of code. Following this, we suggest alternative approaches for the automatic generation of code within the problematising paradigm described above, and highlight some potential applications. We conclude by discussing how automated programming could be guided by modern Computational Creativity practice and in return enhance our philosophical understanding of software being creative, and describe future research directions to explore the creative potential of automated code generation.

Automated Programming Approaches

In common usage, the term 'automatic programming' refers to a range of techniques devised to enable people to program more efficiently, e.g., source code creation through templates within an IDE. Within Artificial Intelligence research, the notion of the fully automated construction of computer programs is to be found within the fields of machine learning, evolutionary programming and automated programming synthesis. We look at each here with respect to their suitability for problematising the world via valued code generation rather than problem solving.

In machine learning, the most obvious areas where automated programming is found are when the learned classifiers are explicitly code, as with Inductive Logic Programming (Muggleton and De Raedt 1994), i.e., where Prolog programs are learned for classification and prediction problems. However, *all* machine learning methods effectively learn representations that are easily interpretable as computer programs. Importantly, deep learning methods are currently being investigated as automated programming systems, with the learned networks examined as computer programs in addition to approximations of neural structures. For instance, deep learning luminary Yann LeCun has recently stated:

> "Deep Learning has outlived its usefulness as a buzzphrase ... Vive Differentiable Programming! ... the important point is that people are now building a new kind of software by assembling networks of parameterized functional blocks and by training them from examples using some form of gradient-based optimization."
> *facebook.com/yann.lecun/posts/10155003011462143*

As an example of the power of deep learning for automated coding, we can turn again to graphics. Previously, in order to transfer an artist's style, or a particular texture, from one image to another – e.g., producing a pastiche of an artist's work by applying a filter to a given image – one had to write a bespoke program or devise a macro in an application like Adobe's Photoshop. The graphics community has investigated style transfer for aspects such as colour (Abadpour and Kasaei 2007), and produced scores of individual style transfer methods for various artists/textures, with pastiche generation finding its way into mainstream graphics packages. However, this was only done for famous artists or particularly useful textures, as hand-programming style transfer methods was time consuming. With the advent of neural network approaches (Gatys, Ecker, and Bethge 2016), a deep neural model can be trained which applies the style of one image to the content of a second image, with impressive results and no user programming required. It is clear that each application of the approach generates a program, albeit in the form of a generative neural network, which performs the same function as the previously hand-crafted ones.

The drawbacks to the usage of machine learning techniques to expose truly unknown unknowns revolve around both the supervised nature of the application and the format of the programs which can be learned. In general, supervised machine learning techniques involve labeling examples into classes and generating methods which can classify unseen examples with high accuracy. As such, supervised methods are suitable for solving known unknowns, where the user knows what he/she wants, but doesn't know exactly what it looks like. However, this is too restrictive to imagine supervised machine learning approaches being used to expose unknown unknowns. Unsupervised methods like data mining can make discoveries the user didn't know they were looking for, but even in these situations, the nature of the discoveries is usually prescribed in advance, in terms of their structure, underlying concepts, or the way they relate variables in the data. Hence, these are not truly unknown unknown problems which are unearthed. If we refer back to the edge detection hypothetical example, that kind of discovery is of software which does something that no-one has thought of doing before, which does not project well onto either supervised or unsupervised machine learning methods.

Another drawback of deep learning techniques for our purposes is that their representation of a "program" is rather opaque from a human perspective. They do not produce code, at least not in a form that a human software engineer would recognise. A trained Artificial Neural Network (ANN) consists of a network structure (which is generally a product of human effort rather than of the AI system), along with thousands or millions of parameter values. Despite efforts to explain and visualise the workings of ANNs (Montavon, Samek, and Müller 2018), it is difficult to appreciate the beauty in a well-tuned ANN in the same way one might appreciate the beauty in a well-written program. For humans at least, it is easier to understand and appreciate a million lines of C++ code than to understand a million real-valued parameters. Notwithstanding efforts in Computational Creativity research to provide alternative scenarios in which creative software can be evaluated, e.g., modeling empowerment for intrinsic motivation (Guckelsberger, Salge, and Colton 2017) or societal curiosity (Saunders 2007), the evaluation of the products and processes of creative systems is largely human-centric. Hence, at least for the time being, it seems that if generated code is to be appreciated culturally, then it should either be understandable by humans (which, as argued below, most generated code is not), or there should be some way of generating high-fidelity explanations of it automatically, noting that different types of users will appreciate code in different ways (Cook et al. 2013).

Evolutionary programming techniques such as genetic programming (GP) produce code directly in a variety of languages, using a sophisticated array of search techniques with crossover and mutation at their heart, guided by user given (or sometimes machine learned) fitness functions. Moreover, they are used for both supervised tasks, e.g., generating classifiers for machine learning applications, and in unsupervised tasks, for instance in evolutionary art projects where the user specifies the fitness of genomes (code) by selecting between phenomes (e.g., images) generated by executing the code (Romero and Machado 2007). The expressivity of many GP approaches means that they could in principle construct code of real value which does something that no one has thought of doing before, potentially exposing unknown unknowns and problematising the world.

Unfortunately, the nature of crossover and mutation does not lend itself to the production of easily understandable code. Overly complicated code is not a problem when the value of the application of the software outweighs the value of the code itself. However, in projects where the code itself is to be celebrated, this issue could be a barrier to usage of GP approaches. In addition, there has been some recognition in the field of Computational Creativity, that the process by which an artefact is created is used in assessing the value of the artefact itself (Colton 2008), and that creative systems should *frame* both their processes and products in order to enable full appreciation of the creative act(s) they perform (Charnley, Pease, and Colton 2012). Artists have embraced the idea of Darwinian-like evolution of software driving artistic projects: they often use scientific descriptions of evolution when framing their art, and present the evolution of their pieces in terms of the family tree of offspring phenomes (e.g., images). However, neither of these is the same as using the actual construction process of a program to add value to the creative act and the product.

In a comprehensive survey, Gulwani et. al (2017) describe automated program synthesis as:

> "... automatically finding a program in the underlying programming language that satisfies the user intent expressed in the form of some specification."

With deep learning being used directly for program synthesis, as described by Balog et al. (2017), it seems likely that there will be a step change in the abilities of software to automatically generate code in this context. However, the situation in automated program synthesis is that – almost without exception – automatic generation of software is done to solve a particular problem in a supervised manner. Problem

types include finding code which can turn given inputs into given outputs and improving existing code, for instance via genetic improvement as per the Gen-O-Fix software (Swan, Epitropakis, and Woodward 2014) or via code transplantation (Barr et al. 2015). While these approaches create new code, they do not cater for the situation where the user has data or existing code that he/she wants to investigate by automatically generating programs, but doesn't know exactly how that investigation should proceed. Indeed, to the best of our knowledge, it seems that no one has ever applied program synthesis in a setting where what constitutes a "good" generated program is unknown in advance and is to be discovered via the process itself. As such, it is difficult to imagine employing the methods developed in this field to generate code that no one knows anything about in advance.

Certain methods and methodologies from Artificial Intelligence research have found natural application in Computational Creativity projects, while others have barely been used. Evolutionary programming, for example, is a mainstay of the field, while machine learning has had fewer, but notable applications, and automated program synthesis approaches have – to the best of our knowledge – never been applied to tasks associated with creative behaviour. There have been successful ad-hoc approaches to direct code generation where, to some extent, the code has been appreciated over and above its value in application (Cook et al. 2013). Here, within the context of the generation of entire videogames by the ANGELINA system (Cook, Colton, and Gow 2016), code was generated directly to control the action of the player's character when they pressed the special powers key (the space bar) on a keyboard. Game levels were then generated around this special character function, i.e., which required the usage of the function in order to complete the level. The generated code was not simply applied, but examined as a valuable artefact, to pass on information to players about what it did (although this was not needed in most cases). Notwithstanding these ad-hoc approaches, in general the idea of creatively generating code as artefacts to be appreciated in their own right has been largely underinvestigated in Computational Creativity research.

Creative Approaches to Automated Programming

Taking the above critique into account, to maximise the potential for automated code generation, we advocate approaches which produce **human-understandable** code in **human-like** ways. By "human-like ways" we mean mirroring the various ways in which human software engineers approach the task of programming: a set of logical iterative steps, top-down or bottom-up, drawing heavily on design patterns and other accepted wisdom, and generally bearing almost no resemblance to the search-based techniques often used by automated systems. In this way, the generated code could be properly appreciated by people, and the creative system can appeal to the code construction method when framing its efforts. We propose two such approaches here.

Given the logical nature of programming languages and the plethora of mathematical/statistical applications of code, it is not surprising that many early (and indeed many modern) computer scientists were originally mathematicians. In this vein, programming languages and logic have much overlap, with Prolog, for instance, being described as a form of logic, a database and a programming language, and logic and maths topics being essential in a computer science education. It is therefore not too difficult to imagine a generative system able to produce mathematical theories graduating from a mathematical discovery system to an automated coding system, which is precisely what we have started to do with the HR program (Colton and Muggleton 2006).

The latest iteration of the software, called HR3 and described by Colton, Ramezani, and Llano (2014), was reengineered from scratch to be an automated programming system, while inheriting the mathematical abilities developed for previous versions. As an example of the difference in approaches, in the domain of number theory, HR2 was given background knowledge including data about which numbers wholly divide which other numbers, e.g., it was given the full set of divisors for the numbers 1 to 1000. It would then invent mathematical concepts, such as perfect squares and prime numbers, and express these in human-readable ways, e.g., as LaTeX sentences or in first order logic (for integration with third party automated theorem provers, model generators and constraint solvers). It would further hypothesise, using standard mathematical symbols, certain conjectures about the concepts which were empirically true, e.g., that an integer can never be both prime and square. Where possible, it would appeal to theorem provers and model generators to prove/disprove the conjectures, and use this to assess and rank the concepts and conjectures, in advance of presentation to the user.

In contrast, HR3 is given as background knowledge Java code which can generate integers up to a user-given limit on the number line, and code which can determine the divisors of a given integer. It then invents concepts in similar ways to HR2, but the concepts are themselves expressed as Java methods which can be run independently of HR3, with the user supplying integer inputs. Conjectures are similarly presented as Java methods which, when executed, test the conjecture empirically over given data (which, as in the case of number theory, can be generated). Hence, if a particular conjecture has tested true on the integers from 1 to a million before being presented, a user can choose to run the method up to 1 billion, before further investigation. The user can, of course, interpret the concepts and conjectures expressed as methods in Java in more mathematical ways if they so desire, but they are spared the common task of implementing them, as they are originally generated directly as code.

In addition to working with background knowledge that is Java code, HR3 can work with standard data from machine learning applications and/or Prolog files. In addition to the code-centric redesign of the software, another important innovation is the usage of *randomised data* to filter uninteresting conjectures. As described by Colton, Ramezani, and Llano (2014), HR3 is orders of magnitude faster than HR2 and is able to generate millions of concepts and conjectures in minutes/hours (depending on the extent of the data over which it is running). The space of Java methods which

HR3 can generate includes numerous duplicates, which on the surface look different, but perform essentially the same calculation. When HR3 makes conjectures relating these methods, they are ultimately disappointing, as they express an artefact of the code construction process rather than a discovery about the data over which the code is run. However, such disappointing conjectures can be discarded if they are also true of randomised data, as the probability of the conjecture arising because of a pattern in random data is miniscule, so the conjectures must relate code not data, and can be discarded (although HR3 can use the information about related code to substitute slower programs with faster ones). A final innovation in HR3 is an ability to make conjectures which are not 100% true, in preparation for working on noisy data.

We note that the HR systems have been used many times to problematise the world, by inventing mathematical concepts and asking questions about their value, then making conjectures and asking questions about their truth. As an early example, HR1 invented the concept of refactorable numbers as those integers for which the number of divisors is itself a divisor. It further hypothesised that odd refactorable numbers are square numbers, which – along with other generated conjectures – was proved by Colton (1999). This discovery was certainly an unknown unknown as the user (first author) had no conception of refactorable numbers in advance: HR1 told him something new and interesting in number theory that he didn't know he was looking for.

Many approaches to automated programming treat the problem as one of search: there exists some notional space of fully-formed programs, and the job of the automated programming system is to locate a particular point in this space. In contrast, programming as a human activity is a highly iterative process, in which a program is built and refined over a period of development. Arguably, this approach bears some resemblance to local search in the program space, but this does not seem to capture the nature of software engineering as an iterative process. There are many formalisms for iterative software engineering by people. An interesting example is test-driven development (TDD) (Beck 2002), which breaks programming into a series of rapid cycles. A new feature (or bug fix) is implemented into a program by (1) writing one or more unit tests to verify the prospective feature; (2) writing the bare minimum of code to cause the new test case to pass without breaking any existing tests; and (3) refactoring the code to add structure and remove duplication. TDD leads to better quality code at the expense of increased development time (George and Williams 2004). TDD could be used to guide automated programming, allowing systems to build a software artefact iteratively rather than monolithically, but to our knowledge, this has not been studied.

The game designer Sid Meier says that "[playing] a game is a series of interesting choices" (Rollings & Morris 2000). This also fits creative processes for artefact generation, so it seems promising to apply decision-making techniques from games in areas where a series of interesting choices must be made, which of course characterises programming. One such technique is Monte Carlo Tree Search (MCTS) (Kocsis and Szepesvári 2006; Chaslot et al. 2006), which is successful in a wide variety of adversarial games (Browne et al. 2012) and decision problems beyond games (Mańdziuk 2018). A closely related technique is Nested Monte Carlo Search (NMCS) (Cazenave 2009), which outperforms MCTS in deterministic single-agent domains. White, Yoo, and Singer (2015) compare MCTS and NMCS to more traditional Genetic Programming approaches. MCTS and NMCS are found to be competitive on some benchmark tests, which were rooted in the idea of automated programming as problem solving. Game tree search, as performed by MCTS approaches is objective-focused, in that it aims to "win the game" (i.e., find a terminal state with a high value according to some utility function). This lends itself more to the problem-solving paradigm of AI than to artefact generation. Methods such as novelty search (Lehman and Stanley 2011) or surprise search (Yannakakis and Liapis 2016), which explore a space for novel or surprising instances rather than seeking a global optimum, may be useful if the aim is to generate interesting programs rather than solve a concrete problem.

Potential Applications of Code Creation

We expect that giving software the ability to creatively generate code will have myriad uses. We propose the following general areas in which code-centric creative automated program generation may be employed with a significant impact.

- Problematising emerging scientific domains

Scientific understanding is constantly being updated in response to new results, which come from new data derived from new experiments, often via new machinery. On the cutting edge of scientific domains, breakthroughs such as improved instrumentation, a theoretical advance, or unexpected experimental results can lead to an explosion of activity when lots of concepts and conjectures are proposed and understanding emerges. As an example, brain scanning equipment occasionally becomes more accurate in sensing structure and activity in brains, i.e., at previously unseen resolutions. In response, physical models of the brain can be challenged and updated and/or new ones invented to capture the information arising from the higher resolution scans.

In these emerging fields, there is as much a need to problematise current understanding as there is to solve problems which have arisen. When people do this, they notice patterns with no explanation, invent concepts to capture groupings without knowing precisely the conceptual definition, pose hypotheses on small amounts of empirical evidence and attempt to find more substantial support, and attempt to derive explanations to phenomena without the necessary language. Often, as the concepts, conjectures and explanations become more concrete, they will be turned into program code to become operationalised, which affords more accurate study of the scientific data being harvested.

We propose to automate the task of problematising scientific understanding from the opposite direction. That is, rather than starting from observations and ending up at code, we suggest using a system such as HR3 to start by automatically inventing code which exposes previously unforseen patterns in data, then conceptualising from the code. In

particular, we initially intend to frame the task of problematising a given dataset as finding quadruples of code $\langle A_1, A_2, A_3, A_4 \rangle$. Here, algorithms A_1 and A_2 manipulate data to produce outputs related by algorithm A_3, and algorithm A_4 shows that this relationship may be interesting. As an example, A_1 and A_2 could output a number for each datum in the dataset, A_3 could relate A_1 and A_2 with a boolean output which is true whenever A_2 produces a multiple of A_3, and A_4 measures the proportion of data points for which this is true. An individual quadruple could be selected and presented because, for the relationship A_3, the output of A_4 was the highest among those with A_3 in the third slot.

The algorithms in any quadruplet which is statistically significantly true of the data can be analysed as an empirically-supported hypothesis, which may lead to more general conceptual definitions, from which more hypotheses will flow, leading ultimately to an improved theoretical understanding of the processes which produced the data, via the lens of the generated algorithms. By enabling the automatic generation of all four algorithms in a quadruple, we hope to maximise the chances of discovering problems that are both interesting and truly unexpected. It may even be feasible and desirable to close the loop on the scientific process and automate the design and execution of experiments to gather more data in response to the discovery of patterns expressed via generated code, as per the *Robot Scientist* described by Sparkes et al. (2010), where machine learning, rather than creative code generation, drove the process.

- Self-modifying automated creators

Many existing creative systems could be enhanced by enabling the software an to alter its own code and/or produce new code for enhanced functionality. We hypothesise that this would lead to increased appearance of autonomy, surprising levels of novelty and more sophisticated processing in the creative systems. As an example, we investigated painting style invention in Colton et al. (2015). This was done with offline generation of different styles as sets of parameters, with online style choices driven by machine vision. An alternative approach would be for the software to invent code to control aspects of the painting process, e.g., how it simulates natural media, and how it uses them in simulated drawing and painting processes. In this context, we can imagine generative software detailing the code generation in commentaries to accompany paintings, framing its creations with descriptions of how it invented new processes – which could have much cultural appeal in the arts.

The FloWr system has been used for process invention, via the generation of novel flowcharts which act as generative text systems, as described by Charnley, Colton, and Llano (2014). As an example output, FloWr invented a flowchart which took the speeches of Churchill as input, extracted phrases with high negative or positive valence, then outputs them in pairs where each has the same footprint (number of syllables). This approach could be enhanced with code invention for the processes *inside* each node of the flowchart. One cultural application of this that we plan to undertake is to produce an anthology of poems, each of which has been generated by a different flowchart, with

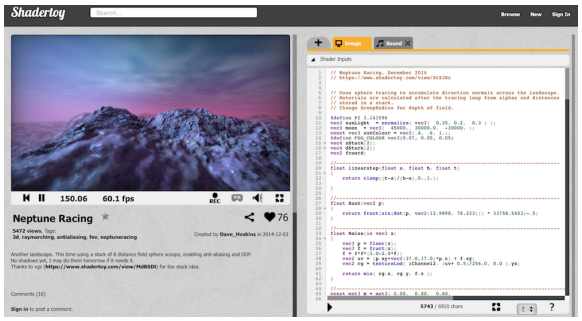

Figure 2: Example ShaderToy post, where users share WebGL shader programs producing 3D animated effects (in this case, a flyover of a procedurally generated landscape).

nodes containing generated code. Here, each poem would be portrayed alongside the code which generated it, so both can be celebrated in the anthology, similar to how Montfort and Fedorova (2013) present both poems and code.

As mentioned previously, Cook et al. (2013) investigated how direct code generation could be used to successfully invent game mechanics for videogames. Here, the ANGELINA system performed code generation as part of the game design process, but this could be taken further, so that games themselves procedurally generate code for game mechanics, in a similar way to how they perform procedural content generation (PCG). Just as PCG keeps games fresh, extends their lifetime and adds intrigue, procedural mechanic generation could do similar, perhaps in a puzzle context, where players have to work out what the game mechanic is doing through usage of it. In this context, we can imagine the game presenting the code for a game mechanic in user-understandable ways, as hints to how best to use it.

- Contributing to recreational coding communities

There are several communities of programmers who write code for recreational, rather than pragmatic, purposes. These have varying levels of application to "real-world" domains. At one end of the scale, the *ShaderToy* community (shadertoy.com), illustrated in figure 2, write GPU fragment shader programs to produce complex 3D visualisations, which have applications in the development of games and other real-time graphics. ShaderToy's roots can be traced back to the *demoscene*, a digital art culture which arose in the 1980s around producing advanced graphical and audio effects from the home computers of the time (Reunanen 2017). Many of the techniques developed by demoscene coders, particularly in the domain of real-time 3D rendering, found applications in games and other software domains.

Much less concerned with real-world applicability are those programmers who write *quines* (Hofstadter 1979), which are programs that output their own source code, and *polyglots*, which are programs whose source code is a valid program in multiple languages, performing the same or different tasks in each. These exercises in creative intellect have arguably no practical purpose, but the activity of programming and reading the programs of others is an interesting pastime to many. An even more extreme example is the *International Obfuscated C Code Contest* (ioccc.org), where

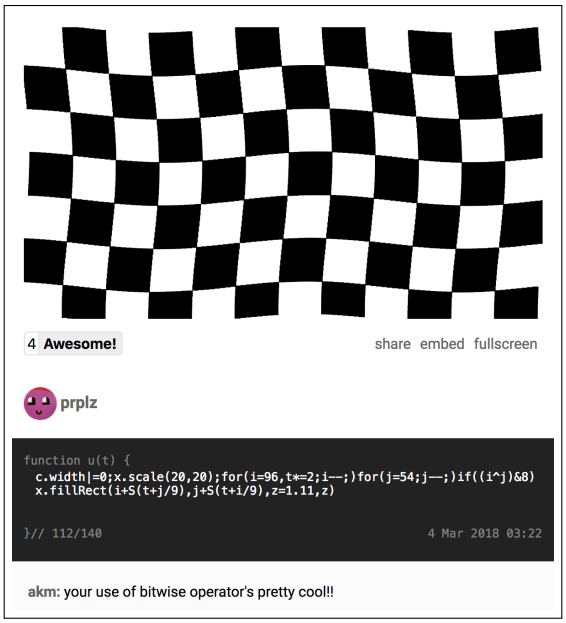

Figure 3: Example Dwitter post (140-character JavaScript program producing animated effects) of a chequered flag blowing in the wind, and follower comment.

participants compete to solve problems in the most unusual and intentionally obtuse ways possible, contrary to the usual principles of good software engineering. Between the two extremes are communities such as *Dwitter* (dwitter.net, see fig. 3), and *Code Golf* (codegolf.stackexchange.com), where programmers write programs (to generate animated graphics and solve various computational problems, respectively), but try to do so with the fewest characters possible.

The appeal of recreational programming is twofold. To the programmer, it is an enjoyable intellectual exercise, like a cryptic crossword or logic puzzle. In this regard, it has similarities to programming-based puzzle games such as *Shenzhen I/O*, *Silicon Zeroes* and *Human Resource Machine*, where players code as part of the gameplay, although these games tend to adapt a more traditional view of coding as problem solving, albeit in a game environment. To the community, it is a celebration of code as an artefact that exists solely for its own sake, to be appreciated in itself, rather than as producing useful/pleasing output. We believe that automated creative programming approaches such as those described above could contribute to recreational programming communities, perhaps producing intriguing software that people might not normally think of coding, in a similar way to how boardgame playing agents occasionally make moves that grandmasters would not necessarily think of.

Conclusions and Future Work

We have argued that Computational Creativity research would be well served by investigating the creative act of software engineering and implementing systems for generative coding. We proposed that such automated software production should lead to culturally appreciated programs which problematise the world, providing enhanced understanding and opportunities in areas like scientific discovery, generative systems and recreational coding. We proposed approaches including extending mathematical theory formation to automated programming, and applying MCTS to making decisions in iterative code formation.

Whatever the method for achieving creative automated programming, if applied for the purposes of problematising the world, this will need to be guided by current thinking in Computational Creativity research. The problem solving paradigm has dominated over the artefact generation paradigm in AI research largely because solving carefully hand crafted problems is guaranteed to be valuable, and systems can be formally evaluated in terms of quality of solution, efficiency, coverage, etc. Notwithstanding some competitions pitching one generative system against another, such as the Mario level generation competition (Shaker et al. 2011), it is generally difficult to compare and contrast the kinds of artistic creations or scientific discoveries that Computational Creativity systems generate. This will be exacerbated when generated code artefacts expose unknown unknowns, as the user should be totally unaware of the problem posed, and hence likely to not recognise its value easily.

In this context, it seems likely that the software will have to make some efforts to convince users of the value of the code it generates, drawing on approaches to framing its processes and products as per Charnley, Pease, and Colton (2012). Moreover, evaluations of generated algorithms may actually involve assessing the creativity of the software which produced them, and could draw on work by Colton, Pease, and Charnley (2011). In return, enabling software to generate code could help solve a philosophical issue which we could term the "mini-me" problem: that creativity is projected onto the programmer and/or the user of generative software because of the explicit nature of the instructions given through human programming. When creative software can in principle rewrite its entire code-base, it will be possible to argue that the programmer has negligible effect on how the software operates or what it produces.

The first steps towards applying the HR3 system to creative code generation have been taken (Colton, Ramezani, and Llano 2014), but there is much work still left to do. In particular, we need to address how to scale up from the software producing small programs to more sophisticated software. We will be investigating both the approach to producing code-quadruples to expose unknown unknowns described above, where four algorithms will be related, and HR3 inventing code to sit inside larger flowcharts generated by the FloWr system (Charnley, Colton, and Llano 2014). We will also look into how sampling methods for both data and the code space can be used to enable HR3 to search deeper for larger algorithmic constructions. Finally, we plan to develop a methodology whereby domain experts such as scientists or game designers can easily provide guidance to the code generation process in terms of mathematical/programmatic/logical ingredients for code and give feedback about the quality of the code produced.

The study of deep learning systems as generators of computer programs hints at an explosion of interest across computer science in software systems programming themselves

and/or automatically designing code for practical and problem solving purposes. Within this ecosystem, there will be space for a range of approaches, including unsupervised approaches to creative automated program synthesis as proposed here. It is possible that deep learning will be seen historically as the first truly successful approach to automated programming, and we hope that more unsupervised approaches, driven by the desire to problematise the world through generated code celebrated in its own right, will also be influential in the progress of computer science.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful suggestions, which have improved this paper. This work has been funded by EC FP7 grant 621403 (GRO).

References

Abadpour, A., and Kasaei, S. 2007. An efficient PCA-based color transfer method. *J. Visual Comm. and Image Representation* 18(1).

Balog, M.; Gaunt, A.; Brockschmidt, M.; Nowozin, S.; and Tarlow, D. 2017. Deepcoder: Learning to write programs. In *Proceedings of the International Conference on Learning Representations*.

Barr, E.; Harman, M.; Jia, Y.; Marginean, A.; and Petke, J. 2015. Automated software transplantation. In *Proceedings of the International Symposium on Software Testing and Analysis*.

Beck, K. 2002. *Test-Driven Development*. Addison-Wesley.

Browne, C.; Powley, E. J.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. on Computational Intelligence and AI in Games* 4(1).

Canny, J. 1986. A computational approach to edge detection. *IEEE Trans. Pattern Analysis and Machine Intelligence* 8(6).

Cazenave, T. 2009. Nested Monte-Carlo Search. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*.

Charnley, J.; Colton, S.; and Llano, T. 2014. The FloWr framework. In *Proc. 5th Int. Conference on Computational Creativity*.

Charnley, J.; Pease, A.; and Colton, S. 2012. On the notion of framing in computational creativity. In *Proceedings of the 3rd ICCC*.

Chaslot, G; Saito, J.; Bouzy, B.; Uiterwijk, J.; and van den Herik, H. 2006. Monte-Carlo Strategies for Computer Go. In *Proceedings of BeNeLux Conference on Artificial Intelligence*, 83–91.

Colton, S., and Muggleton, S. 2006. Mathematical applications of Inductive Logic Programming. *Machine Learning* 64.

Colton, S.; Halskov, J.; Ventura, D.; Gouldstone, I.; Cook, M.; and Perez Férrer, B. 2015. The Painting Fool sees! New projects with the automated painter. In *Proceedings of the 6th International Conference on Computational Creativity*.

Colton, S.; Pease, A.; and Charnley, J. 2011. Computational creativity theory: The FACE and IDEA descriptive models. In *Proc. of the 2nd International Conference on Computational Creativity*.

Colton, S.; Ramezani, R.; and Llano, T. 2014. The HR3 discovery system: Design decisions and implementation details. In *Proceedings of the AISB Symposium on Scientific Discovery*.

Colton, S. 1999. Refactorable numbers - a machine invention. *Journal of Integer Sequences* 2.

Colton, S. 2008. Creativity vs. the perception of creativity in computational systems. In *Proc. AAAI Spring Symp. Creative Systems*.

Cook, M.; Colton, S.; Raad, A.; and Gow, J. 2013. Mechanic miner: Reflection-driven game mechanic discovery and level design. In *Proceedings of the EvoGames Workshop*.

Cook, M.; Colton, S.; and Gow, J. 2016. The ANGELINA videogame design system, parts I and II. *IEEE Transactions on Computational Intelligence and AI in Games* 9(2) and 9(3).

Duff, T. 1988. Re: Explanation, please! `http://www.lysator.liu.se/c/duffs-device.html`.

Gatys, L.; Ecker, A.; and Bethge, M. 2016. Image style transfer using convolutional neural networks. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*.

George, B., and Williams, L. 2004. A structured experiment of test-driven development. *Information and Software Technology* 46.

Guckelsberger, C.; Salge, C.; and Colton, S. 2017. Addressing the 'Why?' in computational creativity. In *Proceedings of the 8th International Conference on Computational Creativity*.

Gulwani, S.; Polozov, O.; and Singh, R. 2017. Program synthesis. *Foundations and Trends in Programming Languages* 4(1-2).

Hofstadter, D. R. 1979. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo Planning. In *Proc. of European Conference on Machine Learning*.

Lehman, J., and Stanley, K. 2011. Abandoning objectives: Evolution through the search for novelty alone. *Evo. Comp.* 19(2).

Mańdziuk, J. 2018. MCTS/UCT in solving real-life problems. In *Advances in Data Analysis with Computational Intelligence Methods*. Springer.

Montavon, G.; Samek, W.; and Müller, K.-R. 2018. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing* 73.

Montfort, N., and Fedorova, N. 2013. Small-scale systems and computational creativity. In *Proceedings of the 4th International Conference on Computational Creativity*.

Muggleton, S., and De Raedt, L. 1994. Inductive Logic Programming: Theory and methods. *Logic Programming* 19-20(2).

Reunanen, M. 2017. *Times of Change in the Demoscene: A Creative Community and its Relationship with Technology*. Ph.D. Dissertation, University of Turku.

Rollings, A., and Morris, D. 2000. *Game Architecture and Design*. Coriolis.

Romero, J., and Machado, P., eds. 2007. *The Art of Artificial Evolution: A Handbook on Evolutionary Art and Music*. Springer.

Saunders, R. 2007. Towards a computational model of creative societies using curious design agents. In *Engineering Societies in the Agents World VII, Vol. 4457 of LNAI, Springer*.

Shaker, N.; Togelius, J.; Yannakakis, G.; Weber, B.; Shimizu, T.; Hashiyama, T.; Sorenson, N.; Pasquier, P.; Mawhorter, P.; Takahashi, P.; Smith, G.; and Baumgarten, R. 2011. The 2010 Mario AI championship: Level generation track. *IEEE TCAIG* 3(4).

Sparkes, A.; Aubrey, W.; Byrne, E.; Clare, A.; Khan, M.; Liakata, M.; Markham, M.; Rowland, J.; Soldatova, L.; Whelan, K.; Young, M.; and King, R. 2010. Towards robot scientists for autonomous scientific discovery. *Automated Experimentation* 2(1).

Swan, J.; Epitropakis, M.; and Woodward, J. Gen-O-Fix: An embeddable framework for dynamic adaptive genetic improvement programming. *Technical report CSM-195, University of Stirling.*

Tatham, S. 2000. Coroutines in C. `https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html`.

White, D. R.; Yoo, S.; and Singer, J. 2015. The programming game: Evaluating MCTS as an alternative to GP for symbolic regression. In *Proc. Genetic and Evo. Computation Conference*.

Yannakakis, G. N., and Liapis, A. 2016. Searching for surprise. In *Proc. 7th International Conference on Computational Creativity*.