

# Assessing Progress in Building Autonomously Creative Systems

Simon Colton\*, Alison Pease†, Joseph Corneli\*, Michael Cook\* and Teresa Llano\*

\*Computational Creativity Group, Department of Computing, Goldsmiths, University of London, UK

†School of Computing, University of Dundee, UK  
ccg.doc.gold.ac.uk

## Abstract

Determining conclusively whether a new version of software creatively exceeds a previous version or a third party system is difficult, yet very important for scientific approaches in Computational Creativity research. We argue that software product and process need to be assessed simultaneously in assessing progress, and we introduce a diagrammatic formalism which exposes various timelines of creative acts in the construction and execution of successive versions of artefact-generating software. The formalism enables estimations of progress or regress from system to system by comparing their diagrams and assessing changes in quality, quantity and variety of creative acts undertaken; audience perception of behaviours; and the quality of artefacts produced. We present a case study in the building of evolutionary art systems, and we use the formalism to highlight various issues in measuring progress in the building of creative systems.

## Introduction

Creativity, we believe, relates to a perception that others have of certain behaviours exhibited by some person or system, rather than an inherent property of people or software: in this sense it is a secondary quality. Moreover, we believe that, just as the endless debates about “is it art?” fuel innovation in the arts, the endless debates about “is it creative?” are a force for good: they drive forward creative practices and Computational Creativity research. A longer discussion of this philosophical position is given in (Colton et al. 2014), and an exposition of creativity as being *essentially contested* (Gallie 1956) is given in (Jordanous 2012).

In such a context of energetic and subjective debate about creativity, it has been difficult to derive systematic approaches to assessing progress in the building of software for creative purposes. One main issue has been the cross-purposes of the creativity project(s) for which software is developed. A useful analogy with the notions of weak and strong AI has arisen recently in Computational Creativity research. Focusing on software which generates artefacts such as poems, paintings or games, we can say that *weak Computational Creativity* objectives emphasise the production of increasingly higher valued artefacts, whereas *strong Computational Creativity* objectives emphasise increasing the perception of creativity people have of the system. This is similar to the distinction put forward in (al-Rifaie and Bishop

2012). In many projects, there are both strong and weak objectives, and often they are not complementary. For instance, increasing autonomy in software may lead simultaneously to higher perception of creativity and lower value artefacts being produced. This is described as the *latent heat* problem in (Colton and Wiggins 2012), and is analogous to U-shaped learning, where to get better, we first have to get worse.

The objectives for a project usually influence the assessment methods employed. In particular, to assess progress with respect to weak objectives, it makes sense to evaluate the quality of the artefacts produced. In contrast, for strong objectives, it makes more sense to assess what software actually does and how and why people perceive it as creative or not. To this end, in (Colton, Pease, and Charnley 2011) we introduced the FACE descriptive model to formalise descriptions of the creative acts undertaken by software, and the IDEA model to formalise the impact those creative acts might have on people. Subsequent attempts to use these models to describe particular systems have highlighted another major issue: the assignment of programmer/software ownership of creative acts. Along with other issues in applying it to describe systems, we have found the FACE model to be inadequate for fully capturing the interplay of creative acts between programmer and program in this respect.

We describe here the next stage of our formalism for capturing notions of progress in building creative systems. We first provide a potted history of how progress has been evaluated in Computational Creativity research, and lay out some intuitive notions of progress. Given our philosophical and practical standpoints, we place less emphasis on asking whether artefacts are ‘better’ than previously. We also avoid direct questions about ‘creativity’ in computational systems. Instead, we integrate (i) aspects of the FACE and IDEA models (ii) objective measures of quality, quantity and variety of creative acts and (iii) audience perceptions of software behaviour and quality of output. We present a two-stage method for estimating whether obvious or potential progress or regress has occurred when building a new system. This involves diagrammatically capturing various timelines in the building and execution of a system, then comparing diagrams. We use the method to describe progress of an evolutionary art system, leading to a general discussion about how the approach could be used in practice. We conclude by describing future directions for this formalism.

## Background in Assessing Creative Progress

The assessment of progress in building creative systems has been a bespoke and multi-faceted endeavour, driven by various, often opposing objectives ranging from understanding human creativity to practical generation of artefacts to the raising of philosophical questions. The majority of practical researchers who engineer and test software joined the Computational Creativity field with objectives in the weak sense of getting software to produce quality artefacts. Hence the first way in which progress was assessed was Boolean: if software reliably produces artefacts of a particular type, then this is progress over software which was unreliable or unable to produce artefacts of the required form.

In such a context, Turing-style discrimination tests indicated a particularly strong milestone: if certain artefacts – usually hand-selected – looked/sounded so like human-authored counterparts that observers couldn't tell the difference, progress had certainly been made. This approach was pioneered by (Pearce and Wiggins 2001) who were one of the first to emphasise the importance and role of evaluation in Computational Creativity, and to propose a concrete way of applying Popperian falsificationism. However, despite them urging caution at depending on the discrimination test to evaluate creativity, direct comparison of human produced and computer generated artefacts has frequently been used to assess progress. We further criticised such Turing-style tests in Computational Creativity, for, among other reasons, encouraging naïvety in software and the generation of pastiches (Pease and Colton 2012). Moreover, we question whether this methodology, while beneficial for short-term scientific progress, is actually detrimental to the longer-term goal of embedding creative software in society (Colton et al. 2014). The work of (Ritchie 2007) was an important step away from simplistic discrimination tests, establishing an approach to assessing the value of artefacts according to their *novelty*, *typicality*, and *quality* within a genre. A number of practitioners have used this approach to compare and contrast their systems, e.g., (Pereira et al. 2005).

As the field matured, attention moved from *mere generation* to programs able to assess, critique and select from their output. Often searching large spaces, software was required to find the best artefacts using mathematically derived or machine-learned aesthetic/utilitarian calculations (Wiggins 2006). If a later version of software – with more sophisticated internal assessment techniques – was able to produce higher yields of higher quality artefacts when assessed externally, then clear progress had been made.

Audience perceptions of software became a focus, as the field further matured. Jordanous used methods from linguistics to determine how people are using the word 'creativity', and which other concepts are associated with it, and then used crowdsourcing techniques to evaluate a creative system in terms of the associated concepts (Jordanous 2012). As a complement to Jordanous's work in which she tried to *capture* society's perception of creativity, researchers began investigating ways to *influence* people's perception of creativity in software. Software assessing its own work made it *appear* more intelligent, and *seem* more creative. This led to the engineering of software that *framed* its processes and

outputs by producing titles, commentaries and other material. (Charnley, Pease, and Colton 2012) propose that this may increase perception of creativity, and audiences would possibly appreciate the artefacts produced more. Studying audience perceptions of creativity in software opened many research avenues, but raised an important problem: that the original product-based assessment methods no longer capture all intuitions of what constitutes progress in the field.

From a strong perspective, some researchers, including ourselves, are not content to accept the underlying assumption of product-based evaluation methods: if better artefacts are produced, the software must have been improved, hence people will project higher perceptions of creativity onto the software and progress will have been made. As mentioned previously, the main problem here is that increasing autonomy – which must happen if strong objectives are to be met – can decrease artefact value. Conversely, when the objectives of a project are weak, it is perfectly natural to decrease software autonomy to produce artefacts of presentation quality, especially when a concert/exhibition is looming, but this is unlikely to increase any perceptions of creativity.

Concentrating on understanding perceptions of software creativity by the general public, we introduced the *creativity tripod* in (Colton 2008b) as three types of behaviours which were necessary (but not necessarily sufficient) for software to avoid being labelled as 'uncreative'. We proposed that people are influenced by their understanding of what software does when assessing its output. We argue that it is easy to ascribe uncreativity to software which is not simultaneously seen as *skillful*, *appreciative* and *imaginative*.

Focusing on assessment of progress by peers, we introduced the FACE and IDEA descriptive models in (Colton, Pease, and Charnley 2011) and (Pease and Colton 2011). The FACE model categorises generative acts by software into those at (g)round level, during which base objects are produced, and (p)rocess level, during which methods for generating base objects are produced. These levels are subdivided by the types of objects/processes they produce:  $F_g$  denotes a generative act producing some framing information,  $A_g$  denotes an act producing an aesthetic measure,  $C_g$  denotes an act producing a concept and  $E_g$  denotes an act producing an example of a concept. Generative acts producing new processes are defined accordingly as  $F_p$ ,  $A_p$ ,  $C_p$  and  $E_p$ . Tuples of generative acts are compiled as *creative acts*, and various calculations and recommendations are suggested in the model with which to compare creative systems. We developed the IDEA model so that creative acts and any impact they might have could be properly separated. We defined various stages of software development and used an ideal audience notion, where people are able to quantify changes in well-being and the cognitive work required to appreciate a creative act and the resulting artefact/process.

We have arrived at a very observer-centric situation in the assessment of progress towards creative systems, in which progress can only be measured using feedback from independent observers about both the quality of artefacts produced and their perceptions of creativity in the software. Unfortunately, the majority of researchers develop software using only themselves as an evaluator, because observer-

based models are too time-consuming to use on a day-to-day progress. These informal in-house evaluation techniques generally do not capture the global aims of the research project, or of the field (e.g. producing culturally important artefacts and/or convincing people that software is acting in a creative fashion). In many cases, systems are presented as feats of engineering, with little or no evaluation at all (Jordanous 2012). We argue that assessing *progress* is inherently a process-based problem. We focus here on modeling diachronic change across multiple levels.

## A Formal Assessment of Progress

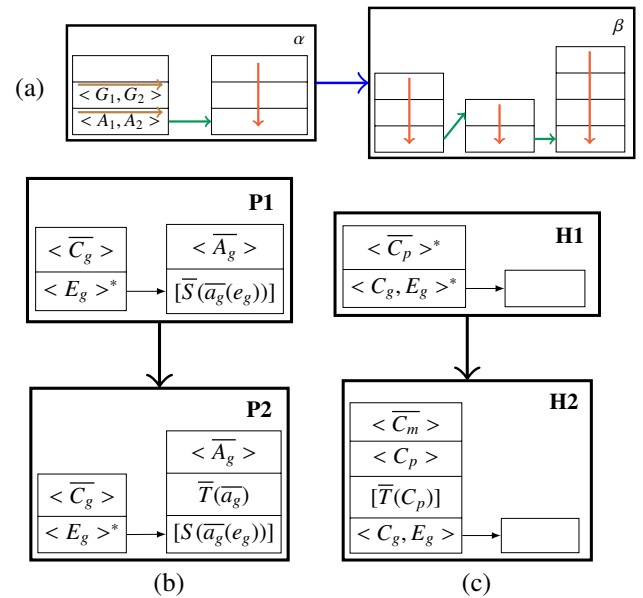
We combine the most useful aspects of the IDEA and FACE models, an enhanced creativity tripod, and aspects of assessing artefact value into a diagrammatic formalism for evaluating progress in the building of creative systems. We focus on the creative acts that software performs, the artefacts it produces and the way in which audiences perceive it and consume its output. We simplify by assuming a development model where a single person or team develops the software, with various major points where the program is sufficiently different for comparisons with previous versions. We aim for the new formalism to be used on a daily basis without audience evaluations, to determine short term progress, but for it also to enable fuller audience-level evaluations at the major development points. We also aim for the formalism to help determine progress in projects where there are both weak and strong objectives. We found that the original FACE model didn't enable us to properly express the process of building and executing generative software. Hence another consideration for our new model is that it can capture various timelines both in the development and the running of software in such a way that it is obvious where the programmer contributed creatively and where the software did likewise.

With the above aims in mind, we envisage a scenario where we are comparing two versions of creative software  $v1$  and  $v2$ . At the highest level, we split the assessment method into a two stage process as follows:

1. Diagrams are drawn for both  $v1$  and  $v2$  which capture the interplay of programmer and program behaviours as timelines during both the development phase and the runtime execution of both versions of the software.
2. The diagrams for  $v1$  and  $v2$  are compared by an audience to determine if the second system represents progress over the first in terms of process. Similarly, the output from  $v1$  and  $v2$  is compared, to see if progress has been made.

### Stage 1: Diagrammatic Capture of Timelines

Taking a realistic but abstracted view of generative software development and deployment, we identify four types of timeline. Firstly, generative programs are developed in **system epochs**, with new versions being regularly signed off. Secondly, each process a program undertakes will have been implemented during a **development period** where creative acts by programmer and program have interplayed.



**Figure 1:** (a) Key showing four types of timelines (b) Progression of a poetry system (c) Progression of the HR system.

Thirdly, at run-time, data will be passed from process to process in series of creative and administrative **subprocesses** performed by software and programmer. Finally, each subprocess will comprise a sequence of generative or administrative **acts**. We capture these timelines diagrammatically, highlighted with coloured arrows in Figure 1(a). The blue arrow from box  $\alpha$  to  $\beta$  represents a change in epoch at system level. The red arrows overlapping a *process stack* represent causal development periods. The green arrows represent data being passed from one subprocess to another at run-time. The brown arrows represent a series of generative/administrative acts which occur within a subprocess.

Inside each subprocess box is either a  $\langle$  creative act  $\rangle$  from the FACE model (i.e., a sequence of generative acts), or an  $[$  administrative act  $]$  which doesn't introduce any new concept, example, aesthetic or framing information/method. Administrative acts were not originally described in the FACE model, but we needed them to describe certain progressions of software. For our purposes here, we use only  $T$  to describe a translation administrative act often involving programming, and  $S$  to describe when an aesthetic measure is used to select the best from a set of artefacts. To add precision, we indicate the output from which generative act the administrative routine is applied, and to which examples a ground aesthetic is applied. To enable this, we employ the FACE model usage of lower-case letters to denote the output from the corresponding upper-case generative acts. We extend the FACE notion of (g)round and (p)rocess level generative acts with (m)eta level acts during which process generation methods are invented. As in the original description of the FACE model, we use bar notation to indicate that a particular act was undertaken by the programmer. We use a superscripted asterisk (\*) to point out repetition.

As a simple example diagram, Figure 1(b) shows the progression from poetry generator version P1 to P2. In the

first version, there are two process stacks, hence the system works in two stages. In the first, the software produces some example poems, and in the second the user chooses one of the poems (to print out, say). The first stack represents two timesteps in development, namely that (a) the programmer had a creative act  $\langle \overline{C}_g \rangle$  whereby he/she came up with a concept in the form of some code to generate poems, and (b) the programmer ran the software to produce poems in creative acts of the form  $\langle E_g \rangle^*$ . The second stack represents the user coming up with an idea for an aesthetic, e.g., much rhyming, in creative act  $\langle \overline{A}_g \rangle$ , and then applying that aesthetic  $\overline{a}_g$  him/herself to the examples,  $e_g$ , produced by the software, in the *selection* administrative act  $[\overline{S}(\overline{a}_g(e_g))]$ , which maps the aesthetic  $\overline{a}_g : \{e_g\} \rightarrow [0, 1]$  over the generated examples, and picks the best one. In the *P2* version of the software, the programmer undertakes *translation* act  $[\overline{T}(\overline{a}_g)]$ , writing code that allows the program to apply the rhyming aesthetic itself, which it does at the bottom of the second stack in box *P2*.

Figure 1(c) shows a progression in the HR automated theory formation system (Colton 2002) which took the software to a meta-level, as described in (Colton 2001). HR operates by applying production rules which invent concepts that categorise and describe input data. Each production rule was invented by the programmer during creative acts of the type  $\langle \overline{C}_p \rangle$ , then at run-time, HR uses the production rules to invent concepts and examples of them in  $\langle C_g, E_g \rangle^*$  acts. In the meta-HR version, during the  $\langle \overline{C}_m \rangle$  creative act, the programmer had the idea of getting HR to form theories about theories, and in doing so, generate concept-invention processes (production rules) in acts of the form  $\langle C_p \rangle$ . The programmer took meta-HR's output and translated  $[\overline{T}(C_p)]$  it into an implemented production rule that HR could use, which it does at the bottom of the stack in box H2.

## Stage 2: Comparing Diagrams and Output

In both simple cases of Figure 1, it is clear that progress has been made in the strong sense, but not clear in the weak sense, as the output could easily be degraded by the more sophisticated processing of the systems. The diagrams help us to capture the creative interplay between software and programmer at design time and run time. However, given that the ultimate aim of both strong and weak projects is to impress audiences with process and product, any assessment of progress must be done in a context of audience evaluation. However, as mentioned previously, audience evaluation is too expensive to help assess progress on a day to day basis. Hence, it seems sensible for the programmer to step in and act as a proxy for a perceived audience: we advocate the programmer putting themselves in the position of the type of person they would expect to form their audience, and answer questions about the products and processes accordingly.

Examining the transition from one diagram to another should provide some shortcuts to estimate audience reactions, especially when there are strong project objectives. In particular, as with the original FACE model, the diagrams make it obvious where creative or administrative responsibility has been handed over to software, namely where an

act which used to be barred has become unbarred, i.e., the same type of generative act still occurs, but it is now performed by software rather than programmer. This happened when the  $\overline{S}$  became an  $S$  in Figure 1(b) and when the  $\overline{C}_p$  became a  $C_p$  in Figure 1(c). At the very least in these cases, an unbiased observer would be expected to project more autonomy onto the software, and so progress in the strong sense has likely happened. In addition, the diagrams make it obvious when software is doing more processing in the sense of having more stacks, bigger stacks or larger tuples of acts in the stack entries. Moreover, the diagrams make it clear that more varied or higher-level creative acts are being performed by the software – again, this was one of the benefits of the original FACE model. Both of these have the potential to convince audience members that software is being more sophisticated with respect to various behaviours described below, and hence can be a shorthand for progress.

When dealing with actual external evaluation, where people don't know what software does, we suggest that the diagrams above (and verbalisations/simplifications of them) can be used to describe to audiences what the software and what the programmer have done in a project. In this way, using also their judgements about the artefacts produced, people can make fully informed decisions in evaluation studies. As a general philosophical standpoint, we suggest not asking people if they believe software is behaving creatively, but rather concentrating on whether they perceive the software as acting *uncreatively*. Our argument for this is that the concept of creativity is essentially contested (Gallie 1956), hence, no matter how sophisticated our software gets, we should not expect consensus on such matters. However, we have found that people agree much more on notions of uncreativity: if a program doesn't exhibit behaviours onto which certain words like *intentionality* can be projected, then it is very easy to condemn it as being uncreative.

Hence, we advocate not asking a set of questions from which we can conclude that an audience member thinks that software is creative, but rather asking questions from which we can determine whether they think that software is acting uncreatively. It may seem like rather a negative admission, but we believe that the best way to get people to accept software as being creative is for them to eventually realise that there is no good reason to call it uncreative. Even then, people would be perfectly at liberty to say that while software is not uncreative, it is not creative either: creativity and uncreativity do not appear to be exact opposites. With this in mind, we have boiled down audience evaluation of behaviour to asking people whether they would project certain words onto software in reaction to understanding what it did in the context of a particular project. We then tentatively conclude that they believe the software is uncreative if they don't project onto it some or all of these words, as originally intended in the creativity tripod proposition (Colton 2008b).

In the five years since the introduction of the creativity tripod, we have slowly added additional behaviours which we have found to be important in the perception of creativity in software. That is, for people to take seriously software as being not uncreative, we believe it needs to exhibit behaviours onto which people can meaningfully project (at least) these

Product change	Process change	Weak	Strong
Up	Up	OP	OP
Up	Down	PP	PR
Up	Same	OP	PP
Down	Up	PR	PP
Down	Down	OR	OR
Down	Same	OR	PR
Same	Up	PP	OP
Same	Down	PR	OR
Same	Same	PP	PP

**Table 1:** Guidelines for using change in evaluation of product and process in gauging (O)bvius or (P)otential (P)rogress or (R)egress, in both weak and strong agendas.

eight words: skill, appreciation, imagination, learning, intentionality, accountability, innovation, subjectivity and reflection. We have found that assessing the level of projection of these words onto the behaviours of software can help us to gauge people’s opinions about (the lack of) important higher-level aspects of software behaviour, such as autonomy, adaptability and self-awareness.

The method we suggest for estimating progress from version  $v1$  of a creative system to version  $v2$  is to: (a) show audience members the diagrams for  $v1$  and  $v2$  as above, and explain the acts undertaken by the software, then (b) show audience members the output from  $v1$  and  $v2$ , and (c) ask each person to compare the pair of product and process for  $v1$  with that of  $v2$ . A statistical analysis could then be used to see whether the audience as a whole evaluates the output as being better, worse or the same, and whether they think that the processing is better, worse or the same in terms of the software seeming less uncreative. This takes into account the phenomenon described in (Colton 2008b) whereby the process can influence value judgements for artefacts.

To use this analysis to estimate progress, it’s important to first prioritise objectives for the project locally in terms of strong and weak agendas. Then, taking the audience evaluation of change in output and in process, we suggest using the guidelines in Table 1. Here, we have stipulated that certain evaluation pairs indicate obvious progression (OP) or obvious regression (OR). For instance, in the weak sense, when the evaluation of output goes up and the evaluation of process increases or stays the same, it seems clear to indicate obvious progress. Other cases are not so clear-cut, for instance when evaluation of artefacts goes up, but evaluation of process goes down. In this case, we suggest that this is potential progress (PP) in a weak agenda, and potential regress (PR) in a strong agenda. In such cases, we give our judgements for whether it is likely, after more development, that  $v2$  will be viewed retrospectively as a progressive success or a step backwards. Note that we have tended to be optimistic, e.g., when evaluation of output and process stay the same, we say that this is potential progress in both weak and strong agendas. Note also that this table is meant to be used flexibly, possibly in a context of more fine grained analysis. For instance, the focus of a subproject might be to increase audience perception of intentionality, and if this increases while audience perception of the value of the process as a whole reduces, it should still be seen as progress.

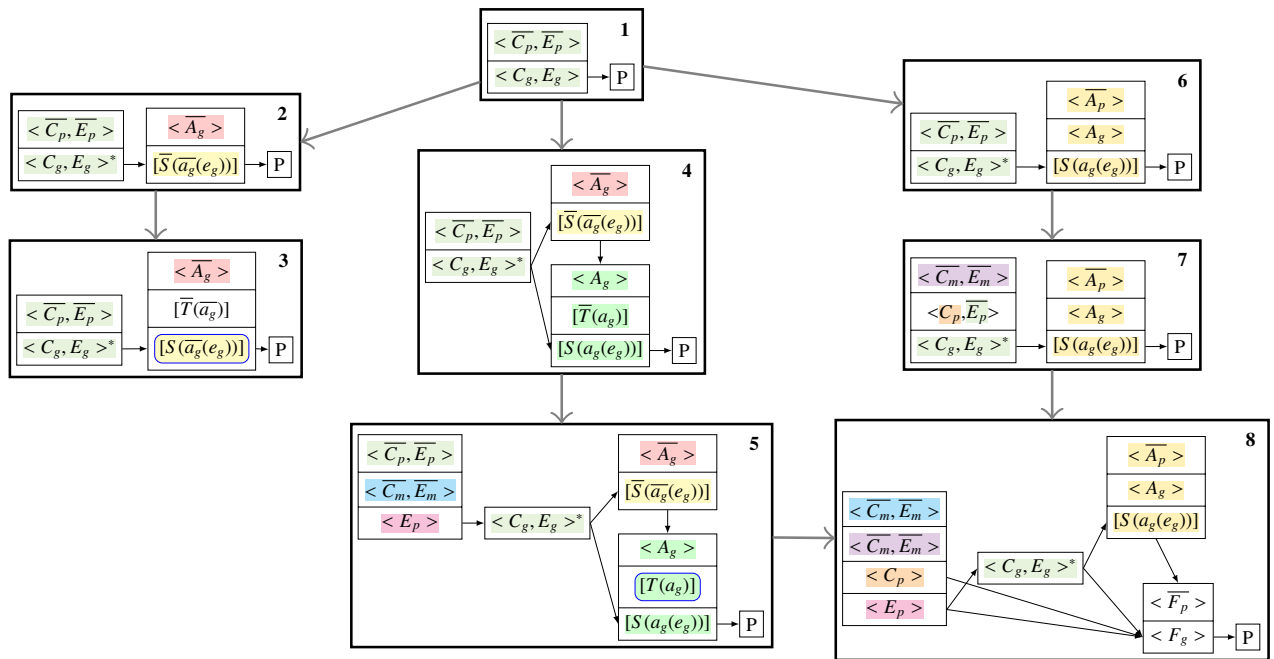
## A Case Study in Evolutionary Art

Evolutionary art – where software is evolved which can generate abstract art – has been much studied within Computational Creativity circles (Romero and Machado 2007). Based on actual projects which we reference, we hypothesise here the various timelines of progress that could lead from a system with barely any autonomy to one with nearly full autonomy. Figure 2 uses our diagrammatic approach to capture three major lines of development, with the final (hypothetical) system in box 8 representing finality, in the strong sense that the software can do very little more creatively in generating abstract art. Since features from earlier system epochs are often present in later ones, we have colour-coded individual creative acts as they are introduced, so the reader can follow their usage through the systems. If an element repeats with a slight variation (such as the removal of a bar), this is highlighted. The figure includes a key, which describes the most important creative and administrative acts in the systems. Elements in the key are indexed with a dot notation: *system.process-stack.subprocess* (by number, from left to right, and top to bottom, respectively). System diagrams have repetitive elements, so that the timelines leading to its construction and what it does at run-time can be read in a stand-alone fashion.

Following the first line of development, system 1 of Figure 2 represents an entry point for many evolutionary art systems: the programmer invents ( $\overline{C_p}$ ) (or borrows) the concept formation process of crossing over sets of mathematical functions to produce offspring sets. He/she also has an idea ( $\overline{E_p}$ ) for a *wrapper* routine which can use such a set of functions to produce images. He/she then uses the program to generate ( $C_g$ ) a set of functions and employ the wrapper to produce ( $E_g$ ) an image which is sent to the (P)rinter. The crossover and subsequent image generation is repeated multiple times in system 2, and then the programmer – who has invented ( $\overline{A_g}$ ) their own aesthetic – chooses a single image to print. In system 3, as in the poetry example above, the programmer translates their aesthetic into code so the program can select images. This is a development similar to that for the NEvAr system (Machado and Cardoso 2002).

Following the second line of development, in system 4, the programmer selects multiple images using his/her own aesthetic preferences, and these become the positives for a machine learning exercise as in (Li et al. 2013). This enables the automatic invention ( $A_g$ ) of an aesthetic function, which the programmer translates by hand ( $\overline{T}(a_g)$ ) from the machine learning system into the software, as in (Colton 2012), so the program can employ the aesthetic without user intervention. In system 5, more automation is added, with the programmer implementing their idea ( $\overline{C_m}$ ) of getting the software to search for wrappers, then implementing this ( $\overline{E_m}$ ), so that the software can invent ( $E_p$ ) new example generation processes for the system.

Following the final line of development, in system 6, we return to aesthetic generation. Here the programmer has the idea ( $\overline{A_p}$ ) of getting software to mathematically invent fitness functions, as we did in (Colton 2008a) for scene generation, using the HR system (Colton 2002) together with The Paint-



ID	Event	Explanation
1.1.1	$\overline{C}_p$	The programmer invents the idea of crossing over two sets of mathematical functions to produce a new set of mathematical functions.
1.1.1	$\overline{E}_p$	The programmer implements a wrapper method that takes a set of mathematical functions and applies them to each $(x, y)$ co-ordinate in an image to produce an RGB colour.
1.1.2	$C_g$	The software generates a new set of functions by crossing over two pairs of functions.
1.1.2	$E_g$	The software applies these functions to the $(x, y)$ co-ordinates of an image, to produce a piece of abstract art.
2.2.1	$\overline{A}_g$	The programmer had in mind a particular aesthetic (symmetry) for the images.
2.2.2	$\overline{S}(\overline{a}_g(e_g))$	The programmer uses his/her aesthetic to select a preferred image for printing.
3.2.2	$\overline{T}(\overline{a}_g)$	The programmer took their aesthetic and turned it into code that can calculate a value for images.
3.2.3	$\overline{S}(\overline{a}_g(e_g))$	The software applies the aesthetic to select one of a set of images produced by crossover and the wrapper.
4.3.1	$A_g$	The software uses machine learning techniques to approximate the programmer's aesthetic.
4.3.2	$\overline{T}(a_g)$	The programmer hand-translates the machine learned aesthetic into code.
4.3.3	$S(a_g(e_g))$	The software applies the new aesthetic to choosing the best image from those produced.
5.1.2	$\overline{C}_m$	The programmer has the idea of getting the software to search through a space of wrapper routines.
5.1.2	$\overline{E}_m$	The programmer implements this idea.
5.1.3	$\overline{E}_p$	The software invents a new wrapper.
5.4.2	$\overline{T}(a_g)$	The software translates the machine-learned aesthetic itself into code.
6.2.1	$\overline{A}_p$	The programmer has the idea of getting the software to invent a mathematical fitness function.
6.2.2	$A_g$	The software invents a novel aesthetic function.
6.2.3	$S(a_g(e_g))$	The software selects the best artefact according to its aesthetic function.
7.1.1	$\overline{C}_m$	The programmer has the idea of getting the software to invent and utilise novel combination techniques for sets of functions, generalising crossover.
7.1.1	$\overline{E}_m$	The programmer implements this idea so that the software can invent new combination techniques.
7.1.2	$\overline{C}_p$	The software invents a novel combination technique.
8.4.1	$\overline{F}_p$	The programmer has the idea of getting the software to produce a commentary on its process and artwork by describing its invention of a new aesthetic, combination method and wrapper.
8.4.2	$F_g$	The software produces a commentary about its process and product.

**Figure 2:** The progression of an evolutionary art program through eight system epochs.

ing Fool (Colton 2012b). In system 7, the programmer realises  $(\overline{C_m})$  that crossover is just one way to combine sets of functions, and gives  $(\overline{E_m})$  the software the ability to search a space of combination methods ( $C_p$ ). The software does this, and uses the existing wrapper to turn the functions into images. System 8 is the end of the line for the development of the software, as it brings together all the innovations of previous systems. The software invents aesthetic functions, innovates with new concept formation methods that combine mathematical functions, and generates new wrappers which turn the functions into images. Finally, the programmer has the idea  $(\overline{F_p})$  of getting the software to write commentaries, as in (Colton, Goodwin, and Veale 2012), about its processing and its results, which it does in generative act  $F_g$ .

Tracking how the system diagrams change can be used to estimate how audiences might evaluate the change in processing of the software, in terms of the extended creativity tripod described above. Intuitively, each system represents progress from the one preceding it, justified as follows:

**1 → 2:**  $\langle C_g, E_g \rangle \rightarrow \langle C_g, E_g \rangle^*$

Simple repetition means that the software has more *skill*, and the introduction of independent user selection shouldn't change perceptions about *autonomy*.

**2 → 3:**  $\overline{S} \rightarrow S$

By reducing user intervention in choosing images, the software should appear to have more *skill* and *autonomy*.

**1 → 4:** Introduction of  $A_g$  and  $S(a_g(e_g))$  acts

Machine learning enables the generation of novel aesthetics (albeit derived from human choices), which should increase perception of *innovation*, *appreciation* and *learning*, involving more varied creative acts.

**4 → 5:** Introduction of an  $E_p$  act,  $\overline{T} \rightarrow T$

Wrapper generation increases variety of creative acts, and may increase perception of *skill* and *imagination*.

**1 → 6:** Introduction of  $A_g$  and  $S(a_g(e_g))$  acts

The software has more variety of creative acts, and the invention and deployment of its own aesthetic – this time, without any programmer intervention – should increase perception of *intentionality* in the software.

**6 → 7:** Introduction of a  $C_p$  act

Changes in the evolutionary processes should increase perceptions of *innovation* and *autonomy*.

**5, 7 → 8:** Introduction of an  $F_g$  act

Framing its work should increase perceptions of *accountability* and *reflection*.

With all strands brought together, the programmer does nothing at run-time and can contribute little more at design time. The software exhibits behaviours onto which we can meaningfully project words like *skill*, *appreciation*, *innovation*, *intentionality*, *reflection*, *accountability* and *learning*, which should raise impressions of *autonomy*, and make it difficult to project *uncreativity* onto the software.

## Discussion

Capturing what programmers and software do creatively over long periods and during complicated program executions is difficult and open to variability. The systems in the above case study could easily have been interpreted and presented differently. In essence, we have provided some tools for presenting software development in terms of creative acts, and suggested a mechanism for turning audience perceptions into estimates of progress. We advise flexible application in both cases. In particular, the difference between potential progress and potential regress is quite subtle. Both mean that it is too early to determine whether progress or regress has been made, and the programmer should proceed with caution: the former suggesting cautious optimism and the latter, cautious pessimism. Practically speaking, the programmer may want to review longer term goals, archive previous versions, and/or clarify research directions.

Our approach is currently more tailored to capturing progress in software behaviour than its output. We would understand some resistance to the approach, particularly from researchers with agendas for Computational Creativity in the weak sense. For example, if product evaluations remain the same, yet processing evaluations go up, this is presumably because the software is performing more sophisticated routines. From a weak perspective, the simpler version of the software clearly has advantages, as it produces the same results in a more understandable way. In certain application domains, for instance mathematical discovery, where aesthetics like truth are of paramount importance, a simpler method for finding a result is usually preferred. While reducing complexity of processing normally requires considerable invention or intervention, unless such invention is done by the software itself, the resulting simplicity would tend to increase perceptions of *uncreativity* in software, regardless (or, indeed, because of) how easy it is to understand what it has done.

Our approach is also more tailored towards capturing progress from version to version of the same software than to comparing different programs. However, we have used the formalism to compare systems in the same application domains, such as mathematical discovery systems AM (Lenat 1976) and HR (Colton 2002), and various poetry and art generators. The comparative approach works somewhat here, because it was possible to compare diagrams meaningfully to suggest where one system would likely be perceived as an improvement over the other. However, full application of the approach may be difficult as the context for evaluating artefacts (and the processes producing them) can change greatly with small changes in artefact composition. For instance, we recently attempted to compare one-line “What if...?” ideas produced textually by three systems. We found that it was not possible to conceive a fair approach involving an audience to determine which system's artefacts or processes were the best. Fields like Machine Learning have largely homogenised the testing of their systems in a problem-solving paradigm. Given the tacit requirements for software to surprise us through its output and processing, and to innovate on many levels, it seems unlikely that such standardisation could apply in Computational Creativity research.

## Related Work

Diagrammatic approaches to software modelling have been extensively studied in the last two decades. The best known example is the Unified Modelling Language (UML), managed by the Object Management Group (OMG), a standard that is widely used to visualize the design of systems ([www.omg.org/spec](http://www.omg.org/spec)). The main objectives of modelling with UML are to represent the architecture of a system, including use cases, deployment, information flow diagrams, etc., and to model system behaviour and data flow via activity diagrams, state machines, sequence diagrams, etc.

Progress at the process level can be modelled with UML by diagramming the steps used to complete a task within the system. However, UML is not typically applied to model progress at the level of system epochs, although two UML diagrams can of course be compared on the basis of the functionality they describe. Some diagrams created using the UML model, such as use case diagrams, enable designers to specify the agents that participate in the development of a system: people, external processes, other systems and the system itself can all be modelled as agents. However, there is no formal notation to distinguish between the different agents, rather, they are simply assigned a label which is meaningful for the system designer. The OMG have also developed other graphical notations specialised for other aspects of systems modelling. For instance, the Business Process Model and Notation (BPMN) is used to model business processes by extending the original activity diagrams of UML. The specific objective of BPMN is to provide a high-level overview of business systems, rather than detailed information about how the system works.

UML diagrams have also been used in the context of formal methods. In particular, the UML-B language (Said, Butler and Snook 2009) enables the modelling of Event-B specifications as UML-like diagrams. Event-B is a formalism based on set theory for the modelling and verification of systems (Abrial 2010). One of the main aspects of Event-B is the use of refinement to handle the complexity of systems at different levels of abstraction. UML-B can be used to diagrammatically model a system at increasing levels of refinement, and system consistency can then be verified through mathematical proof. However, UML-B considers one system at a time, so it is not possible to use this formalism to model creative change as system development progresses.

Using the Event-B formalism, it is possible to model aspects of the environment, such as external systems that affect the behaviour of the modelled system. The aim is to ensure that the designed system will work in harmony with its operating environment. However, there is no clear way to delimit the aspects of the model that are related to the environment and those that are part of the final system. Again, the environment is simply identified by the designer assigning meaningful names to the state representing it. Other related approaches include Z-notation (Spivey 1992), the Vienna Development Method (Jones 1990) and the B-method (Abrial 1996). The objective of these approaches is to verify properties of systems. Progress would be meaningful at the modelling level, i.e., by building models that offer increasing detail (and assurance) about how a given system works.

Petri nets provide a graphical notation used primarily to model systems with concurrency (Girault and Valk 2003). With petri nets, progress at the process level is modelled in the form of state transitions, and data is represented by abstract tokens, with no data values assigned. An extension, called coloured petri nets (Jensen and Kristensen 2009), allow data values to be assigned to tokens. Neither type of petri net is used for modelling changes through versions of a system. Petri nets are an event-based modelling language and representations of agents (such as the programmer or the system) are not included in the formalism.

## Conclusions and Future Work

We have presented a new diagrammatic formalism for assessing progress in building creative systems. Our aims were to enable more precise understanding of progress in Computational Creativity in general, and in mapping the progress of particular systems. In doing so, we aimed to bring closer together public/peer appreciation of progress, strong/weak agendas, and day-to-day/milestone progress assessments. The new approach involves producing diagrams of systems that depict creative acts in timelines, which are compared in a context of audience evaluation of process and product. When applied, the formalism captures some intuitive notions, including: quality of artefacts; quantity, level and variety of creative acts performed; and audience perception of software behaviour. To enable better understanding of process, and more informed audience judgements about (un)creativity, the diagrams explicitly separate creative acts coming from the programmer and the program. Even in the absence of audience participation, the diagrams themselves can be used in combination with straightforward assumptions about audience reactions to system design features to perform low-cost estimates of progress in a strong agenda.

We motivated the approach throughout with various philosophical standpoints, as per (Colton et al. 2014), supported by a critical review of the ways in which progress in building creative systems has been measured historically. To highlight the potential for the formalism, we presented a case study where the progress through eight versions of evolutionary art software was mapped and justified.

Our audience evaluation model is far from complete. We plan to employ the criteria specified in (Ritchie 2007), for more fine-grained evaluations of the quality, novelty and typicality of artefacts. We will also import audience reflection evaluation schemes from the IDEA descriptive model, e.g., change in well-being, cognitive effort and emotional responses such as surprise and amusement. We have so far used the diagrammatic approach to fully depict timelines in the building of generative software producing mathematics, visual art, poetry and video games, including dozens of system diagrams (omitted for space reasons). This has worked well, but there are still some subtle improvements required to capture better the functioning of the software at run-time.

(Gabriel and Goldman 2000) describe system development environments with many contributing programmers, and multiple interacting, self-programming, and self-updating distributed systems (Gabriel and Goldman 2006). It would be straightforward to modify our formalism to deal



with multiple agents, for example by turning bars into superscripts. However, this does complicate the notion of progress: if system  $\mu$  chooses to hand off creative control to system  $\nu$ , this would amount to changing a superscript – but it's not immediately clear that this should count as progress in the same way that removing bars does. If the agents are considered to be full partners in the creative process,  $\mu$  and  $\nu$  may well have their own perspectives on what counts as progress, and this needs to be formalized.

Broadly speaking, we expect that the distinction between strong and weak agendas will eventually disappear: in order to produce higher quality artefacts, more sophisticated systems involving behaviours perceived as creative will be required, and audiences will expect to project notions of creativity onto software to fully appreciate its output. In such a context, assessing processes and products simultaneously will be important, and we hope versions of this diagrammatic approach will enable this. In (Colton, Goodwin, and Veale 2012), we used the FACE model as a driving force for poetry generation software, rather than as a descriptive tool. We hope that system developers will similarly begin to think about their software in the above diagrammatic terms, in order to suggest interesting new avenues for implementation.

### Acknowledgements

This work has been supported by EPSRC grants EP/J004049 and EP/L00206X, and through EC funding for the project COINVENT 611553 by FP7, the ICT theme, and the Future Emerging Technologies FET programme. We would like to thank the anonymous reviewers for their helpful comments.

### References

- Abrial, J.-R. 1996 *The B-Book – Assigning Programs to Meanings*. Cambridge University Press.
- Abrial, J.-R. 2010. *Modeling in Event-B – System and Software Engineering*. Cambridge University Press.
- al-Rifaie, M and Bishop, M. 2012. Weak vs. strong Computational Creativity, computing, philosophy and the question of bio-machine hybrids. In *Proceedings of the AISB Symposium on Computing and Philosophy*.
- Charnley, J.; Pease, A.; and Colton, S. 2012. On the notion of framing in Computational Creativity. In *Proceedings of the International Conference on Computational Creativity*.
- Colton, S., and Wiggins, G. 2012. Computational Creativity: The final frontier? In *Proceedings of the European Conference on AI*.
- Colton, S.; Cook, M.; Hepworth, R.; and Pease, A. 2014. On acid drops and teardrops: Observer issues in Computational Creativity. In *Proceedings of the AISB Symposium on AI and Philosophy*.
- Colton, S.; Goodwin, J.; and Veale, T. 2012. Full-FACE poetry generation. In *Proceedings of the International Conference on Computational Creativity*.
- Colton, S.; Pease, A.; and Charnley, J. 2011. Computational Creativity Theory: The FACE and IDEA descriptive models. In *Proceedings of the Int. Conference on Computational Creativity*.
- Colton, S. 2001. Experiments in meta-theory formation. In *Proceedings of the AISB'01 Symposium on AI and Creativity in Arts and Science*.
- Colton, S. 2002. *Automated Theory Formation in Pure Mathematics*. Springer.
- Colton, S. 2008a. Automatic invention of fitness functions with application to scene generation. In *Proceedings of EvoMusArt*.
- Colton, S. 2008b. Creativity versus the perception of creativity in computational systems. In *Proceedings of the AAAI Spring Symposium on Creative Intelligent Systems*.
- Colton, S. 2012. Evolving a library of artistic scene descriptors. In *Proceedings of EvoMusArt*.
- Colton, S. 2012b. The Painting Fool: Stories from building an automated painter. In McCormack, J., and d'Inverno, M., eds., *Computers and Creativity*. Springer.
- Gabriel, R. P. and Goldman, R. 2000. Mob software: The erotic life of code. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.
- Gabriel, R. P. and Goldman R. 2006. Conscientious software. In *ACM SIGPLAN Notices*, 41.
- Gallie, W. 1956. Essentially contested concepts. *Proceedings of the Aristotelian Society* 56.
- Girault, C. and Valk, R. 2003. *Petri nets for systems engineering – a guide to modeling, verification, and applications*. Springer.
- Jensen, K. and Kristensen, L. M. 2009. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer.
- Jones, C. B. 1990. *Systematic software development using VDM*. Prentice Hall.
- Jordanous, A. 2012. A Standardised Procedure for Evaluating Creative Systems: Computational Creativity Evaluation Based on What it is to be Creative. *Cognitive Computation* 4(3).
- Jordanous, A. 2012. *Evaluating Computational Creativity: A Standardised Procedure for Evaluating Creative Systems and its Application*. Ph.D. Dissertation, University of Sussex.
- Lenat, D. 1976. *AM: An Artificial Intelligence approach to discovery in mathematics*. Ph.D. Dissertation, Stanford University.
- Li, Y.; Hu, C.; Minku, L.; and Zuo, H. 2013. Learning aesthetic judgements in evolutionary art systems. *GPEM* 14(3).
- Machado, P., and Cardoso, A. 2002. All the truth about NEvAr. *Applied Intelligence* 16(2).
- Pearce, M. T. and Wiggins, G. A. 2001. Towards a Framework for the Evaluation of Machine Composition. In *Proceedings of the AISB'01 Symposium on AI and Creativity in Arts and Science*.
- Pease, A., and Colton, S. 2011. Computational Creativity Theory: Inspirations behind the FACE and IDEA models. In *Proceedings of the International Conference on Computational Creativity*.
- Pease, A., and Colton, S. 2012. On impact and evaluation in Computational Creativity: A discussion of the Turing test and an alternative proposal. In *Proceedings of the AISB symposium on AI and Philosophy*.
- Pereira, F.; Gervás, P.; and Cardoso, A. 2005. Experiments with assessment of creative systems: An application of Ritchie's criteria. In *Proceedings of the IJCAI Computational Creativity Workshop*.
- Ritchie, G. 2007. Some empirical criteria for attributing creativity to a computer program. *Minds and Machines* 17.
- Romero, J., and Machado, P., eds. 2007. *The Art of Artificial Evolution: A Handbook on Evolutionary Art and Music*. Springer.
- Wiggins, G. A. 2006. Searching for computational creativity. *New Generation Computing* 24(3).
- Said, M. Y., Butler, M. J., and Snook, C. F. 2009. Language and tool support for class and state machine refinement in UML-B. In *Proceedings of the 2nd World Congress on Formal Methods*.
- Spivey, M. 1992. *The Z notation: A reference manual*. Prentice Hall.