The FloWr Framework: Automated Flowchart Construction, Optimisation and Alteration for Creative Systems

John Charnley, Simon Colton and Maria Teresa Llano
Computational Creativity Group, Department of Computing,
Goldsmiths, University of London, UK
ccg.doc.gold.ac.uk

Abstract

We describe the FloWr framework for implementing creative systems as scripts over processes and manipulated visually as flowcharts. FloWr has been specifically developed to be able to automatically optimise, alter and ultimately generate novel flowcharts, thus innovating at process level. We describe the fundamental architecture of the framework and provide examples of creative systems which have been implemented in FloWr. Via some preliminary experimentation, we demonstrate how FloWr can optimise a given system for efficiency and yield, alter input parameters to increase unexpectedness, and build novel generative systems automatically.

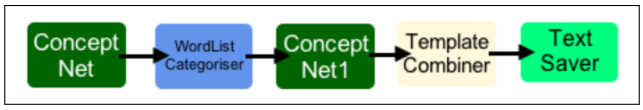
Introduction

One of the main reasons people give for why software should not be considered creative is because it follows explicit instructions supplied by a programmer. One way to reduce such criticisms is to get software to write software, because if a program writes its own instructions, or the code of another program, some level of creative responsibility has clearly been handed over. Automated programming techniques such as genetic programming have been used in creativity projects, such as evolutionary art (Romero and Machado 2007), and software innovating at process (algorithmic) level has been studied in this context. Moreover, machine learning approaches such as inductive logic programming (Muggleton 1991) clearly perform automated programming. In both these cases, programs are generated for specific purposes. In contrast, we are interested here in how software can innovate at process level for exploratory purposes, i.e., where the aim is to invent a new process for a new purpose, rather than for a given task.

Getting software to write code directly is a long-term goal, and we have performed some early work towards this with the invention of game mechanics at code level (Cook et al. 2013). Such code generation will likely be organised at module level, so it seems sensible to study how programs can be constructed in formalisms such as flowcharts over given code modules, to study creative process generation. Flowcharts are used extensively for visualising algorithms, e.g., UML is a standard for representing code at class level (Rumbaugh, Jacobson, and Booch 2004). There are also a handful of systems which allow flowcharts to be developed and automatically converted into code. These include the MSDN VPL (msdn.microsoft.com/bb483088.aspx), the RAPTOR system (Carlisle et al. 2004), and IBM's WebSphere, which allows programmers to visualise the interaction between nodes and produce fully-functional systems on a variety of platforms (ibm.com/software/uk/websphere). Also, *Visual Programming* systems such as Blockly, (code.google.com/p/blockly), AppInventor (appinventor.mit.edu) and Scratch (scratch.mit.edu) allow the structure of a program to be described by using different types of blocks.

We could certainly have investigated process-level innovation by implementing software to automatically control the flowcharting systems mentioned above. However, these systems have been developed to support human-centric program design, and we have had many difficult experiences in the past where we have wrestled unsuccessfully with programmatic interfaces to such frameworks. In addition, in line with usual software engineering paradigms, there is an emphasis on being able to explicitly specify what programs do and an expectation of perfect reliability in the execution of those programs. We are more interested in a flowcharting system able to be given vague instructions (or indeed, none at all) and with some level of automation, produce valuable, efficient flowcharts for generative purposes. For these reasons, we decided to build the FloWr (Flo)wchart (Wr)iter system from scratch with a clear emphasis on automated optimisation, alteration and construction of systems. This paper describes the first release of this framework.

In the next section, we describe the fundamentals of the framework: how programs are represented as scripts which can be created and manipulated visually as flowcharts, and how developers can follow an interface to introduce new code modules to the system. Following this, we detail a FloWr flowchart for poetry generation which uses Twitter, and we use this in an investigation of flowchart robustness. We then present some preliminary experiments to test the viability of FloWr automating various aspects of flowchart design. In particular, we investigate ways in which it can alter and optimise given flowcharts, and we describe an experiment where FloWr invented novel flowcharts from scratch. Notwithstanding a truly huge search space, we show there is much promise for process-level innovation with this approach, and we conclude with a discussion of future research and implementation work.




```
text.retrievers.ConceptNet.ConceptNet_0
dataFile:simple_concept_net_1p0_sorted.csv
relation:IsA
rhsQuery:animal
minScore:0
#wordsOfType = answers[*]

...WordListCategoriser.WordListCategoriser_0
wordList:child;human;apple;
stringsToCategorise:#wordsOfType
#filteredFacts = textsWithoutWord[*]

text.retrievers.ConceptNet.ConceptNet_1
dataFile:simple_concept_net_1p0_sorted.csv
lhsQueries:#filteredFacts
relation:CapableOf
minScore:0
#propertyFacts = facts[*]

...TemplateCombiner.TemplateCombiner_0
templateText:
What if there was a little c1Texts[*][0]
who couldn't c1Texts[*][2]?
numRequired:1000
c1Texts:#propertyFacts
#whatifs1 = instantiatedTemplates[r5]

utility.saving.TextSaver.TextSaver_0
dir:/Output/Flow/whatifs
textsToSave:#whatifs1
```

Figure 1: Ideation script and corresponding flowchart

The FloWr Framework

We aim to use the FloWr framework to investigate automatic process generation via the combination of code modules. As discussed in the subsections below, our approach has been to implement a number of such code modules, which we call *ProcessNodes*, engineer an environment where *scripts* direct the flow of data from module to module, and develop a graphical user *interface* (GUI) to enable visual combination of ProcessNodes into scripts using a flowcharting metaphor.

Individual ProcessNodes

Focusing on generative language systems, we have implemented a *repository* of 39 ProcessNodes for a variety of tasks, from the generation of new material, to text retrieval, to analytical and administrative tasks. For instance, in the repository, there is a ProcessNode for downloading tweets from Twitter, one for performing sentiment analysis, and one for simply outputting text to a file. A new node must extend the Java ProcessNode base class, by implementing its abstract *process* method, which will be called whenever the module is executed. The developer can write whatever software they see fit in the node, and this may call external code in any language. The developer can specify certain *input parameters* for the process, as public fields of the class, along with an optional list of allowed or default values for each parameter. As mentioned below, the scripting mechanism enables variables to be specified, which hold output from processes, and can be substituted in as the input parameters of other nodes. This facilitates the flow of data. At runtime, using Java's reflection mechanism, FloWr will set each ProcessNode's parameters according to the current state of processing, i.e., explicit assignments of the current value of variables to input parameters, prior to calling the *process* method for the node.

The ProcessNode superclass provides a number of utility methods that a node developer can use during processing, such as determining the local location of the *data directory* which holds non-code resources. There are also methods for reporting processing errors during runtime, which developers can use to neatly handle exceptions and other failures. The *process* method of each ProcessNode returns an object of type *ProcessOutput* which holds all the output from the node, hence developers create a Java class that extends the ProcessOutput base class. This facilitates internal FloWr functionality for determining the types of output variables and checking whether a script specifies passing objects of the right type from one node to another (again using Java reflection). Developers can use bespoke or existing classes as fields within output classes, so they can create more complex data-structures for node output. Developers should be aware, however, that most nodes take as input primitive types such as integers and strings, and collections of these, so if they want the output from their nodes to be used by others, their ProcessOutput classes will probably need to have fields at some level in a standard format.

A Scripting Mechanism

A FloWr system is a collection of task-specific ProcessNodes, with a description of how data from each node is selected as input to others, expressed using a script syntax. An example script, which has been edited a little to improve clarity, is given in figure 1. The functionality of this script is described in the subsection on automated optimisation. Each paragraph of the script describes a ProcessNode by specifying its type, configuration and output. The first line is the *type* of node, which refers to the Java class called when that node is run in a system. In figure 1, the first process uses the *ConceptNet* class, from the *text.retrievers* package. Suffixes are used to differentiate between multiple instances of the same node type in a script. When a script is parsed, each type must be an instantiable compiled subclass of ProcessNode in the stated package, which must also contain a valid ProcessOutput subclass.

The next lines in the paragraph specify how the input parameters should be initialised at runtime as name:value pairs. The *name* indicates the parameter to be initialised, and the *value* can be either a simple assignment, or a variable representing some output from another node. Script parsing checks that each name refers to a publicly accessible field of the ProcessNode class, which can be validly assigned with the specified value or the value of the variable indicated. Default parameter assignments are used where a parameter value is blank. Node developers can define any

parameters, so they could develop a single node that operates with various input types, to build more robust systems.

Variable definitions are a **#**-prefixed alphanumeric label and an *output specifier* for a particular part of the output from a process. As mentioned above, each ProcessNode class must have an associated ProcessOutput class. The output specifier refers to the fields defined within this output class, which will be populated by the node at runtime. In its simplest form, the specifier indicates a particular field to assign to the variable. Alternatively, they can be separated by dots, where each segment is a field relative to the specifier to its left. Where the indicated field is a list, square brackets are used to indicate a *selection specifier*, which identifies a subset of elements to be assigned to the variable. The acceptable selection specifiers are: *****: all elements; **f*n***: the first *n* elements; **l*n***: the last *n* elements; **m*n***: the middle *n* elements; and **r*n***: *n* randomly chosen elements.

When a script is run, all processes are checked for syntax errors and data-type inconsistencies. FloWr determines the process run order by inspecting dependencies between output variables and input parameters, and errors are raised whenever there are problematic loops in a script. FloWr then steps through each node in the run order by instantiating an appropriate ProcessNode object, assigning its parameters according to the script, calling its *process* method to execute the node, and storing the output. In the example script of figure 1, we see that the *ConceptNet_0* ProcessNode has output with an `answers` field, which is a list. The whole list (indicated by `answers[*]`) is assigned to the variable `#wordsOfType`, which is passed into the *WordListCategoriser_0* ProcessNode as the input parameter `stringsToCategorise`. In this simple script, each node except the last one assigns a single aspect of its output to a variable, which is passed onto the next node.

A Flowcharting Interface

The FloWr GUI shown in figure 2 is the primary system development tool, where flowcharts are used to visually represent the interaction between ProcessNodes. The interface has several components. Firstly, the central panel displays the flowchart currently being worked upon, with individual ProcessNodes shown as boxes and the arrows between them indicating the transfer of data. The flowchart in figure 2 – the functionality of which is described in the next section – has 16 nodes of 13 different types, with colour coding indicating nodes of the same type or which perform similar tasks. For instance, blue boxes in figure 2 represent ProcessNodes which categorise texts (using word sense, sentiment, regular expressions and a user-supplied word list). To add a new node to a flowchart, the user right clicks the main panel and chooses from a series of popup menus. As might be expected, flowchart boxes can be dragged, resized, deleted, copied and renamed, and multiple boxes in sub-charts can be selected, moved, resized and deleted simultaneously.

When a box is clicked, it gains a thick grey border, and the arrows going into/out of it gain circles, which when clicked populates the *mappings* (upper) internal frame in the GUI with the output variables and input parameters of the two ProcessNodes joined by the arrow. The mappings between nodes can be edited by hand, and arrows are automatically generated whenever an output variable is used as an input parameter for another node. Clicking on a box populates the mappings frame with the input variables and output parameters for that ProcessNode, and populates the *output* (lower) internal frame with the values of the output variables, if they have previously been calculated via a run of the system. In figure 2, we see that the user previously selected the output for the SentimentSorter node, (which is a poem about being abusive) in the output frame. They then selected the circle on the arrow between two nodes, and the output variables and input parameters for LineSplitter and SentimentSorter were displayed accordingly in the mappings frame.

A small black panel containing a play and stop button for executing and halting the script is shown at the top right of the flowchart panel. When the user has chosen to execute the flowchart multiple times from a menu, a number indicating which run is executing is shown in this panel (the number 17 in figure 2). The user can double click a node, and FloWr will run all the processes leading into that node, including it, but not nodes which occur later in the script run order. When the flowchart is running, the node which is actually executing is given a red border: in figure 2 the TextRankKeyphraseExtractor node is running. Nodes can take some time to finish executing, and it is often useful for their output, and the output for all the nodes earlier in the flowchart, to be frozen, i.e., calculated once and stored rather than generated when that process is run again. This can be done using the interface and is indicated with a pushpin in the flowchart box: in figure 2, the Twitter node has been frozen. The pushpins in the mappings frame and the output frame can be used to stop their context from changing when boxes on the flowchart are clicked.

An Example FloWr System

We have used FloWr to hand-craft a number of systems, including flowcharts for poetry generation in a manner similar to that of (Colton, Goodwin, and Veale 2012), where newspaper articles were manipulated to produce poems. We have also used FloWr to perform automated theory formation using the same production rule-based method employed by the HR system (Colton 2002), and we have re-implemented aspects of The Painting Fool software (Colton 2012). Finally, as discussed in the next section, we have used FloWr scripts to produce fictional ideas, with experiments using this given in (Llano et al. 2014a) and (Llano et al. 2014b). In each of these instances, we have developed a fully-operational system and the FloWr GUI has enabled a clear visualisation of the overall system, enabling us to design, edit and tweak each implementation. The ProcessNodes required and the flowcharts implementing these systems are available in the FloWr distribution.

The flowchart in figure 2 produces poems as a collection of related tweets from Twitter in a relatively sophisticated way. Execution begins with a *Dictionary* ProcessNode which selects all the 5,722 words from a standard dictionary with a frequency of between 90% and 95%, with word frequency determined using the Kilgarriff database (Kilgarriff 1997), which was mined from the British National

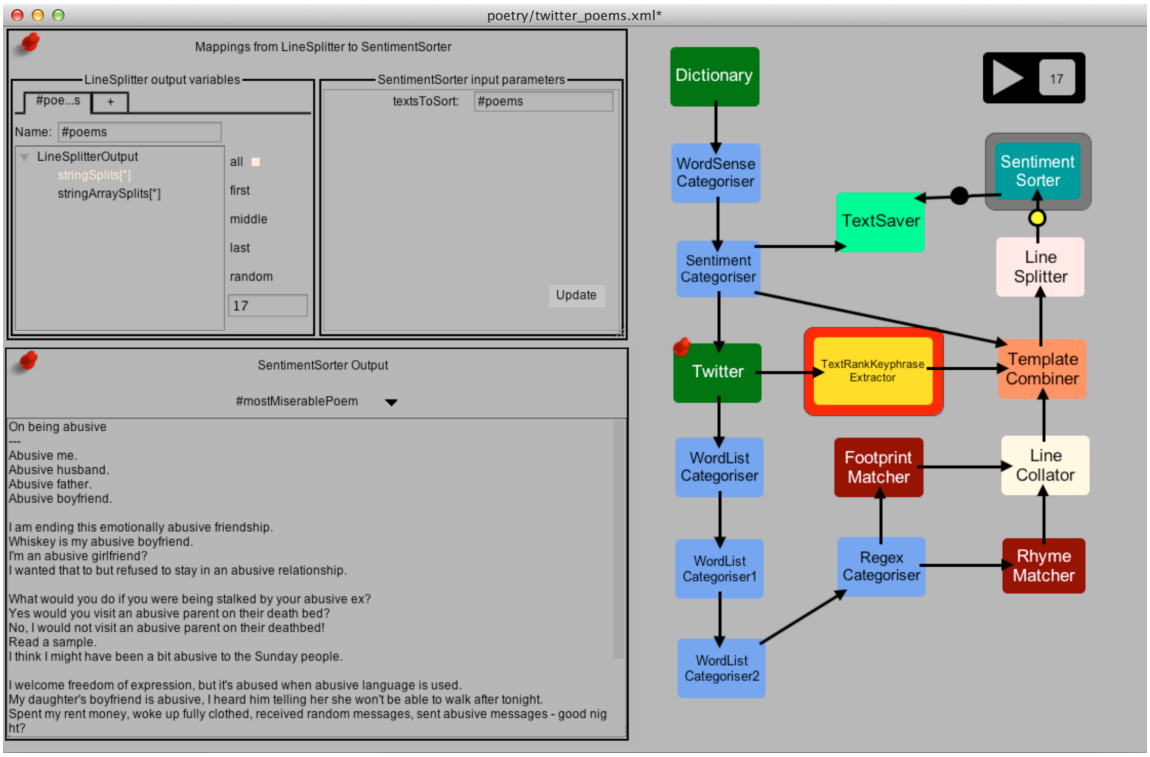


Figure 2: The FloWr flowcharting graphical user interface.

Corpus. Such words are relatively common but not too common or too uncommon in the language. Next in the flowchart, a *WordSenseCategoriser* selects the 772 words that are adjectives (in terms of their main sense) as per the British National Corpus tagset (Leech, Garside, and Bryant 1994). A *SentimentCategoriser* node then splits the adjectives into categories based upon how positive or negative a word is, using the Afinn sentiment dictionary (`fnielsen.posterous.com/tag/afinn`) expanded by adding synonyms from WordNet. From the list of 211 *negative words*, i.e., scoring -1 or less for valency, a single word is randomly chosen as the poem theme, using the variable selection syntax [r1] in the underlying script, as described above.

The *Twitter* ProcessNode accesses the Twitter web service through the Twitter4J library (`twitter4j.org`), and retrieves a maximum of 1,000 tweets containing the theme word – there may be less if the word is not mentioned in many recent tweets. Tweets are cached to make retrieval quicker later on. Also, as part of the retrieval process, the tweets are filtered to remove copies and tweets containing a word which cannot be pronounced, as per the CMU pronunciation dictionary (CPD, at `www.speech.cs.cmu.edu/cgi-bin/cmudict`), or which cannot be parsed using the Twokenize tokenizer (`bitbucket.org/jasonbaldridge/twokenize`). We have found that the 90-95% word frequency previously mentioned ensures that there are usually sufficient tweets (counted in the hundreds) after the filtering process, but that the tweets tend to be less banal than usual, as the usage of a somewhat uncommon word requires some thought.

The retrieved tweets are used in two ways. Firstly, a *TextRankKeyphraseExtractor* node extracts keyphrases using an implementation (`lit.csci.unt.edu`) of the TextRank algorithm (Mihalcea and Tarau 2004) over the entirety of the tweets collated as a paragraph of text. As an example, the poem theme in the run presented in figure 2 was 'abusive', and the keyphrases of 'abusive husband', 'abusive father' and 'abusive boyfriend' were extracted. Secondly, the tweets are passed through a triplet of *WordListCategoriser* nodes which are used to exclude tweets containing undesired words. The first filter removes tweets containing any of a pre-defined list of first names, discarding many tweets about particular people, which are too specialised for our purposes. The second removes tweets containing Twitter-related words such as *retweet*, and the third removes tweets containing certain profanities. The *RegexCategoriser* ProcessNode then splits the tweets into two sets based upon whether or not they contain personal pronouns (I, we, they, him, her, etc.). Only tweets containing personal pronouns are kept, which helps remove commercial service announcements, which are dull. In the 'abusive' example, from the 1000 tweets retrieved, 110 were removed as duplicates or for being unpronounceable/non-tokenisable. 80 were further removed for including first names, 33 for including Twitter terms, 22 for including profanities, and 262 were removed because they included no personal pronouns, leaving 493 tweets for the construction of stanzas in the poem.

The remaining tweets are processed by a *RhymeMatcher* node which finds all pairs of tweets with the same two phonemes at the end, when parsed by the CPD. The num-

On Being Eerie

Eerie me.
Eerie feeling.
Bit eerie.

I hate the basement level of buildings.
You always lose reception and it's always quiet and eerie.
This doesn't quite capture the eerie pink glow of this morning.
Is pop culture satanic?
In a spiritual (not religious) sense?
I don't really know.
But man, there are some eerie parallels.
It's concerning.
I find it very eerie when someone is tinkering with your teeth and telling jokes.
Or is that just me?

I hate winter and the cold, but I love how silent the night is during cold winter weather.
It's eerie, but peaceful.
Old school!
It was always eerie.
No one around, and completely quiet.
It's like being on the wrong end of the apocalypse.
Experiencing the eerie light of total eclipse.
I'm going through it today.
The fact that I'm talking about my grandma in a past tense is eerie and weird to me.

I saw weird stuff in that place last night.
Weird, strange, sick, twisted, eerie, godless, evil stuff.
And I want in.
Yes - that is quite an eerie sound!
It's so eerie listening to the crying in the background.
I can understand that.
It just feels eerie to have it haunt you (word-for-word) by different users.

I hope the cloud stayed away for you.
Wow, how was the eerie darkness?
I thought I told you.
Oops.
Weird, eerie, strange portraits and locations.
An antique metal ship and a candle make for eerie (and awesome) decorations.
I mean the art direction is eerie.
I'm pretty sure it's hogwash.

Bit eerie.
Eerie feeling.
Eerie me.

Figure 3: Example Twitter poem: *On Being Eerie*.

ber of matching phonemes can be changed to increase or decrease the amount of rhyming. From these, 250 pairs are randomly chosen (or the entire set, if less than 250). The tweets are likewise processed by the *FootprintMatcher* node, which counts the number of syllables, again using the CPD, and finds all pairs of tweets with the same footprint. As before, 250 pairs of tweets are chosen randomly. Next, *LineCollator* constructs sets of 16 different tweets in quadruples of the form ABBA, where the As are a pair with equal footprints and the Bs are a pair which rhyme. An example quadruple is as follows (note the two central lines rhyme, and the outer lines both have 17 syllables):

I hope the cloud stayed away for you. Wow, how was the eerie darkness?
I thought I told you. Oops. Weird, eerie, strange portraits and locations.
An antique metal ship and a candle make for eerie (and awesome) decorations.
I mean the art direction is eerie. I'm pretty sure it's hogwash.

The *TemplateCombiner* node brings all the processed information together into a poem based upon a specified poem template. The inputs to this process are the theme word which becomes part of the poem title, the keyphrases which provide a context at the top and (reversed) at the bottom of the poem, and the quadruples from *LineCollator*, which each form a stanza of the poem. *TemplateCombiner* is told to produce 20 poems by choosing 20 sets of quadruples from *LineCollator* randomly. The *LineSplitter* ProcessNode takes each poem and splits any line where there is a period (tweets often contain two or more sentences), which tends to make the poems more *poem-shaped*. Finally, the *SentimentSorter* node selects the poem with the most negative affect, which is saved to a file by the *TextSaver* ProcessNode. This is given the theme word as an input, and the file is so named.

Freq(%)	Structure	Neg.	Stanzas	Yield(%)
85-90	$FRRF$	false	4	94
90-95	$FRRF$	true	4	94
80-85	$FRRF$	false	4	90
90-95	$R_1R_2R_2R_1$	false	4	80
90-95	$R_1R_2R_2R_1$	true	4	74
90-95	$R_1R_2R_2R_1$	false	6	46
90-95	$R_1R_2R_2R_1$	true	6	12

Table 1: Yield results for Twitter poetry flowchart.

In general, we have found that these Twitter poems are surprising and interesting. In particular, the slight rhyming in the centre of the poems is noticeable, and the multiple voices expressed through 16 different tweets, coupled with the often rushed nature of the tweets can give the poems a very dynamic feel. Another example poem is given in figure 3, where the theme was 'eerie'. This poem was recited as part of a poetry evening during a festival of Computational Creativity, in Paris in July 2013 (Colton and Ventura 2014).

The nature of the flowchart, including the ProcessNodes, the I/O connections and the parameterisation of the processes was carefully specified and tweaked by hand over many hours, to produce a poem most of the time, for different adjectives. One of the benefits of the flowcharting approach is that variations can be easily tried out – but it would be frustrating if the yield of poems wasn't consistent. To investigate the robustness of the flowchart, we varied the word frequency parameters in the *Dictionary* to test the retrieval of tweets containing less common words. We also made the poem construction more difficult. Firstly, we introduced all-rhyming stanzas $(R_1R_2R_2R_1)$ rather than the footprint-rhyming structure $(FRRF)$. Secondly, we introduced an additional *SentimentCategoriser* node to ensure that only tweets with an average (Neg)ative valency were used. Finally, we increased the number of stanzas from 4 to 6. For each of 7 setups given in table 1, we provide the yield produced from 50 runs of the flowchart. We note that the flowchart is fairly robust to lowering the theme word frequencies, but the volume of tweets didn't support well the construction of more complex poems. In fact, only 12% of runs resulted in a poem when six $R_1R_2R_2R_1$ stanzas with only negative tweets were sought. This indicates that there is a limit to how far a successful flowchart can be tweaked before it loses its utility.

Automation Experiments

We present here some preliminary experiments to automatically alter, optimise and generate flowcharts. As mentioned previously, a driving force for the project is to study the potential of automated process generation. FloWr simplifies the process of constructing a system but, as highlighted in the previous section, fine-tuning a chart can be a laborious process. For example, the flowchart/script in figure 1 was developed by hand for a project where the ConceptNet database of internet-mined facts (Liu and Singh 2004) was used for fictional ideation in the context of Disney cartoon characters, as described in (Llano et al. 2014a). Given a *theme* word like 'animal', the flowchart uses the *ConceptNet1* node to find all Xs for which there is a fact [X,IsA,animal], removes spurious results, such as [my_husband,IsA,animal] with the *WordListCategoriser*, and then for a given relation, R, finds all the facts of the form [X,R,Y], using the *ConceptNet2* node. To produce the fictional idea, it inverts the reality of each fact using the *TemplateCombiner* node to produce an evocative textual rendering. For instance, the fact that [cat,Desires,milk] becomes "What if there was a little cat who was afraid of milk?".

In further testing, we substituted 'animal' for other theme words such as 'machine', and produced ideas such as: "What if there was a little toaster who couldn't find the kitchen" (by inverting the LocatedNear relation in this case). There are 49 ConceptNet relations and a large number of couplings of these with theme words, many of which yielded no results. For instance, we found no facts about types of machines and the Desires relation, presumably as machine don't tend to desire things. Focusing on animals, it took around 2 hours to produce the first working flowchart which produced a non-zero yield of facts which could be usefully inverted for the invention of Disney characters. One of the benefits of automation we foresee would be a substantial reduction in this type of manual fine-tuning.

Flowcharts can be constructed and altered in several ways. ProcessNodes can be added, removed or replaced with alternatives. Parameterisations of nodes and the links between them can be amended by modifying, creating or deleting variables and changing input settings. The space of all possible constructions and alterations is vast and, at this early stage, we have restricted ourselves to a subset. Specifically, we have considered changes to parameterisations of existing flowcharts and we describe some experiments in the following section, followed by how these can be guided to achieve particular optimisation objectives. After this, we consider constructing flowcharts from scratch by sequentially adding additional ProcessNodes. In all cases, FloWr has generated flowcharts representing novel and interesting creative tasks whilst avoiding an element of manual construction effort.

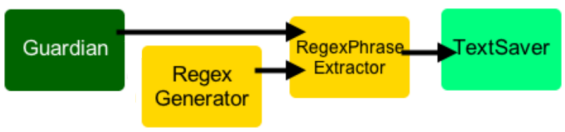


Figure 4: Flowchart for automated regex generation.

S	NW	FWLen	WLCh	FLCh	LLCh	Yield(%)	Av.
1	3	3-6	equal	equal	none	55	48.6
2	3	3-6	equal	any	none	42	12.1
3	3-5	3-6	any	any	none	24	9.24
4	3	3-6	incr.	incr.	none	38	5.1
5	3-5	3-6	any	any	any	0	0

Table 2: Regex generation test yields (tongue twister texts).

Flowchart Alteration

When motivating the building of the FloWr framework in the introduction, we noted that we want the approach to produce unexpected results, with FloWr scripts being somewhat unpredictable. One way to increase unexpectedness is to randomly alter input parameters to ProcessNodes at run-time. We investigated this via the generation of simple tongue-twister texts, by extracting word sequences using regular expressions. We implemented a *RegexGenerator* ProcessNode which produces regular expressions (regexes) such as:

\bs[a-zA-Z]4\b\s1,\bs[a-zA-Z]5\b\s1,\bs[a-zA-Z]6\b

When applied to a corpus of text, this regex extracts all triples of words of length 5, 6 and 7 which begin with the letter 's'. We applied this to a corpus of 100,000 Guardian newspaper articles, and it returned 21 triples, such as 'small screen success' and 'short skirts showing'.

The input parameters to *RegexGenerator* specify the number of words (NW) in the phrases sought, what the first word's length (FWLen) should be, and how the word lengths should change (WLCh): either increasing, decreasing, staying the same, or no(ne) change. The parameters also enable the specification of the first letter of the first word, and how subsequent first letters should change lexicographically (FLCh): increasing, decreasing, staying the same or no(ne) change. The last letter changes (LLCh) can similarly be specified. Importantly, FloWr can be instructed to choose each parameter randomly from a given range. For start and end letters, this range is a-z, for word length and letter changes it is {increase, decrease, equal, none} and the integer value for NW and FWLen can be specified to be within a user-given range.

We implemented the flowchart in figure 4 to input the whole Guardian corpus and a generated regular expression in the *RegexPhraseExtractor* node, and output the resulting text (if any) to a file. We ran five sessions with different input parameter ranges for the *RegexGenerator* node. For each session, we specified that the first letter of the first word should be chosen randomly. In each session, we ran the flowchart 100 times and recorded the yield as the percentage of times when text was actually produced. We also recorded the average number of lines of text produced (i.e., the average number of hits for the regular expression in the corpus). The results are given in table 2. We see that (S)etup 5 is completely unconstrained and the space which is randomly sampled from is dense in poor regexes which have no hits in the corpus: the yield is zero. However, with some constraining of the regex ranges allowed, the yield increases almost to 50%. Also, as expected, the average number of

hits increased in line with the yield. The following are two tongue twisters found in the results for setups 1 and 4:

posted pretax profit	cancer despite everyone
please please please	classy devices emerging
petrol prices played	carbon dioxide expelled
profit public policy	carbon dioxide emission
poorer people pushed	choice defense everyone

In other experiments, with the ideation flowchart of figure 1, we looked at automatically changing the theme word. To do this, a *WordNet* ProcessNode was used to find hypernyms of *animal*, which returned the words *organism* and *being*. We then requested the hyponyms of each of these, which generated 87 alternative themes, which were substituted for the theme in the flowchart. Several of the themes produced a high yield of invertible facts, with 13 theme/relation combinations generating more facts than the highest found by hand. Three of these used theme word *person*, e.g., with the *CapableOf* relation, which generated 2,154 ideas such as the concept of actors being able to face an audience. Similarly, the theme words *individual* and *plant* had high yields. However, one word that was identified automatically using this method was *flora*, which gave interesting invertible facts about trees, such as being homes for nesting birds and squirrels. These were not considered in our manual experiments using the *plant* theme. In a similar way, we used *ConceptNet* to find theme words by inspecting all the *IsA* relations in its database, from which it identified 11,000 themes. Using these, we found the highest yield with the theme *mammal* and the relation *NotDesires*, which we hadn't found manually. This generated 568 facts, mainly about people, e.g., the ideas that people don't want to be eaten or bankrupt, both of which led to interesting fictional inversions.

Optimising Flowcharts

We performed some experiments in automating the task of finding high-yield configurations for the ideation flowchart of figure 1. To do this, we provided a list of themes and asked FloWr to consider all possible pairings of theme and ConceptNet relation. To assess the yield of a ProcessNode, FloWr uses Java reflection to traverse the structure of its output object and count the objects and sub-objects in individual fields or in lists. We have found this to be a reliable measure of output quantity, particularly when assessing relative sizes. It is also general, and will produce a useful yield measure irrespective of the nature of the node and its output. The manual process identified the theme word *animal* and the relationship *CapableOf* as producing the highest yield of 530 usable facts. The automated approach also identified this combination, but it highlighted a more productive relationship for *animal*, namely *LocatedAt*, which provides 1,010 facts. This combination had been overlooked during the manual process, in favour of using the *LocatedNear* relationship, which produced only 39 facts.

We also investigated optimising flowcharts for efficiency. Given a target time reduction and minimum output level for ProcessNodes in a given flowchart, we investigated an approach which identifies small local changes to input parameters that have the most global impact on the system. Firstly, the nodes are ordered according to their increasing contribution to the overall execution time. Considering the slowest ProcessNode, P, first, an attempt is made to establish if the time taken is a consequence of the amount of data it receives, by halving the data given and comparing execution times. If input data is causing P's slow speed, the ProcessNode(s) which produced that input into P are re-prioritised higher than P in the ordering. Moreover, a local goal for each ProcessNode is assigned, which is either to reduce its execution time or the size of its output. Then, local reconfigurations consider incremental changes to numeric and optional parameters until the local goals, or failure, have been met. Any successful local reconfigurations are then applied to the global system and reported to the user if they achieve the overall goals. Multiple tests are used at each stage to confirm that the reported results are consistent.

We successfully applied this approach to the Twitter poetry generator, where it identified that the high average base execution time of 10 seconds was caused by the *WordListCategoriser* nodes processing a high number of tweets from the *Twitter* node. It applied an iterative process, which reduced the *numRequired* parameter by a given percentage for a pre-defined number of steps, noting each time that the node output yield was reduced, eventually settling on a *numRequired* setting of 63. It then tested this on the global system and found that this reduced the overall runtime to 630 milliseconds, whilst still successfully generating poems. In a similar experiment, we optimised another poetry system which used Guardian newspaper articles as source material, as in (Colton, Goodwin, and Veale 2012). The optimisation method found that one node could be optimised by reducing its input size, which led to the altering of another node's input parameters, and a 40% reduction in overall execution time, while the flowchart still produced poems.

Flowchart Construction

We have investigated how to construct FloWr systems from scratch. Working in the context of producing poetic couplets, we tested a method which could generate a system with three to five nodes taken from these sets respectively: {*Twitter, Guardian, TextReader*}, {*WordSenseCategoriser, SentimentCategoriser*}, {*TextRankKeyphraseExtractor, RegexPhraseExtractor*}, {*WordSenseCategoriser, SentimentCategoriser*}, {*FootprintMatcher, RhymeMatcher*}. We used our experience of which nodes work well together to create this structure, and to specify a number of possible options for the input parameters. For some nodes, we were restrictive, e.g., we specified that the *Guardian* node should use a specific date range for selecting articles and always return the same number. For other nodes, we allowed FloWr to use any of the parameter values from the optional lists provided by the node developer. For the *Twitter* node, we chose five dictionary words randomly for queries and *TextReader* was directed to use a set of Winston Churchill speech texts.

Despite these limitations, there are still a huge number of possible combinations to explore. For example, there are 108 possible node combinations, 27,000 parameter combinations and over 261 million variable definition combina-

Figure 5: An automatically generated rhyming couplet system.

tions. The size of this restricted subset makes a brute-force approach intractable, given that many nodes have execution times of over a second. Hence, we tried a depth-first search of all possible systems, by choosing node combinations randomly and configuring each node with input parameters chosen from those allowed at random. Next, the method considers the possible data links between nodes by considering each pair in turn. The set of variables that could be defined in the scripting syntax for the earlier node in the system is compared with the input parameters for the following node. Only those where the output variable type and the input parameter types match will be syntactically valid, and these are chosen from randomly and applied to the script.

We generated 200 scripts using this process and tested each to see whether it produced output from the final node. We found that 17 (8.5%) worked successfully and produced poetic couplets. Of these, 8 contained 3 nodes, 8 contained 4 nodes and one – shown in figure 5 – contained 5 nodes. This script takes Guardian articles from the first week of 2012, extracts the neutral texts in terms of sentiment, and identifies all their key phrases. It selects keyphrases beginning with an adjective and outputs pairs of phrases with the same syllable footprint, producing these:

actual bodily harm metropolitan police	chief inspector working domestic violence	dangerous driving potential recruits

The yield from the 17 scripts varied widely from one to over 4 million couplets. The most commonly used ProcessNode in the successful flowcharts was *TextRankKeyphraseExtractor*, which was used 28% of the time, followed by FootprintMatcher, used 23% of the time. *FootprintMatcher* is more prevalent than *RhymeMatcher* at 5%, because there are more pairs of phrases with the same number of syllables than pairs which rhyme. The *RegexPhraseExtractor* fails to appear, due to limited input data, i.e., there were no strings satisfying the regular expressions sought, due to the limited amount of text available. We experimented with further restricting the types of nodes that could be selected. In particular, using information about the frequency of nodes in successful scripts from the first experiment, we managed to improve the yield of working scripts to 18.5% by allowing only *WordSenseCategoriser* nodes to be used for categorisation.

One particular (four-node) script caught our eye. It takes Churchill texts, extracts keyphrases, keeps only those where the first word has *extreme* sentiment, i.e., ≥ 2 or ≤ -2, then outputs pairs with the same footprint, such as: [great air battle:despairing men] and [greater efforts:greater ordeals]. The 52 poetic couplets that this script generated provided the starting point for a poem written by a collaborator: Russell Clark selected a subset of these pairs, then combined and ordered them into a piece entitled *Churchill's War*, which is shown in figure 6. The poem was one of four submitted for analysis by poetry experts as part of a BBC Radio 4 piece on Computational Creativity (Cox 2014), although a different poem was ultimately read out and analysed.

Churchill's War

Good many people, great differences
good many people: outstanding increase.
Great organisations, greater security
greater security: terrible position

Great combatants, brilliant actions
Great preponderance, greater efforts

Great air battle, despairing men
Great air battle, brilliant actions

Great Britain, good account
Great Britain, good reason

Great flow: Great war
Great flow: Good men

Chess proceeds, good reason
Chess proceeds: victory

Figure 6: A poem based upon the output from an automatically generated process for poetic couplet generation.

Conclusions and Future Work

The FloWr framework enables fairly rapid prototyping of flowcharts for creative systems. We presented here fundamental details of how code modules can be implemented and combined via scripts using a flowcharting front end. We presented flowcharts for producing poems, fictional ideas, tongue twisters and poetic couplets, which re-use nodes for retrieving, categorising, sorting, combining and analysing text. We have performed some experimentation to assess the potential for automating aspects of flowchart design, both to help users construct, vary and optimise flowcharts, and to highlight the potential for FloWr to automatically construct novel processes. The ultimate aim of this project is to provide an environment which encourages third party ProcessNode and flowchart developers to contribute material from which FloWr can learn good practice for innovating in automatic process design. We have already started implementing functionality which enables FloWr to learn flowchart configurations which are likely to produce results. This has aspects in common with other knowledge-

based system design projects, such as *Rebuilder* (Gomes et al. 2005). Ultimately, FloWr will reside on a server, constantly generating, testing and running novel system configurations in reaction to people uploading new ProcessNodes and scripts. We intend to have a large number of nodes covering a variety of different individual tasks in many domains. For instance, we have a variety of NLP nodes, e.g., for Porter Stemming (Porter 1980) and we will be extending this to cover nodes for other tasks, such as tagging and chunking.

The first release of the FloWr framework, along with dozens of ProcessNodes and numerous flowcharts is available at `ccg.doc.gold.ac.uk/research/flowr`. In future releases, we plan a number of improvements to the underlying framework, including much more automation in the system, given the promise shown for this in the experiments described here. The systems that can be implemented currently are quite limited, and we plan to introduce additional programmatic constructs, such as framework level control of looping, and ProcessNode level control of conditionals. We will also implement useful functions, such as FloWr running a sub-flowchart repeatedly until it produces a particular yield for the rest of the flowchart, and translating variables, e.g., from `ArrayList<String>` to `String[]`, to increase flexibility. We will test different search techniques to tame the vast space of flowchart configurations, so that FloWr can reliably generate interesting novel flowcharts, and we will implement the optimisation and alteration routines we have experimented with as default functionalities. We also plan to implement more entire systems in FloWr, in particular we expect The Painting Fool art program (Colton 2012) to eventually exist as a series of flowcharts in FloWr. Also, we have started to port the HR3 automated theory formation system (Colton 2014) to FloWr. We have experimented with HR3 to add adaptability to the Twitter poetry generation flowchart: using concept formation over a given set of tweets, HR3 can successfully find a linguistic pattern which links subsets of tweets, that can be extracted and turned into poem stanzas.

The flowchart in figure 2 is a creation in its own right. To some extent, the value of such flowcharts exists over and above the quality of the output they produce. That is, the way in which the flowchart constructs artefacts is an interesting subject in its own right. For reasons of improving autonomy, intentionality and innovation in computational systems, we believe that software which writes software – whether at code-level or via useful abstractions such as flowcharts – should be a major focus in Computational Creativity research. Automated programming has been adopted, albeit in restricted ways, in highly successful areas of AI such as machine learning, and we believe there will be major benefits for the building of creative systems through the modelling of how to write software creatively.

Acknowledgments

This work has been supported by EPSRC Grant EP/J004049/1 (Computational Creativity Theory), and EC FP7 Grant 611560 (WHIM). We would like to thank Russell Clark for his help with the poetry generation flowcharts and curating their output. We would also like to thank the anonymous reviewers for their helpful comments.

References

Carlisle, M.; Wilson, T.; Humphries, J.; and Hadfield, S. 2004. RAPTOR: Introducing programming to non-majors with flowcharts. *Journal of Computing Sciences in Colleges* 19(4).

Colton, S.; Goodwin, J.; and Veale, T. 2012. Full-FACE poetry generation. In *Proceedings of the International Conference on Computational Creativity*.

Colton, S. 2002. *Automated Theory Formation in Pure Mathematics*. Springer.

Colton, S. 2012. The Painting Fool: Stories from building an automated painter. In McCormack, J., and d'Inverno, M., eds., *Computers and Creativity*. Springer.

Colton, S. 2014. The HR3 discovery system. In *Proceedings of the AISB symposium on computational scientific discovery*.

Colton, S. and Ventura, D. 2014. You Can't Know my Mind: A Festival of Computational Creativity. In *Late Breaking Proceedings of the International Conference on Computational Creativity*.

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*.

Cox, T. (Presenter) Can a Computer Write Shakespeare? BBC Radio 4 documentary, first aired on 15th May 2014.

Gomes, P.; Pereira, F.; Paiva, P.; Seco, N.; Carreiro, P.; Ferreira, J.; and Bento, C. 2005. Rebuilder: a case-based reasoning approach to knowledge management in software design. *Engineering Intelligent Systems for Electrical Engineering & Communications* 13(4).

Kilgarriff, A. 1997. Putting frequencies in the dictionary. *International Journal of Lexicography* 10(2).

Leech, G.; Garside, R.; and Bryant, M. 1994. CLAWS4: The tagging of the British National Corpus. In *Proceedings of the 15th COLING*.

Liu, H., and Singh, P. 2004. Commonsense reasoning in and over natural language. In *Proceedings of the 8th International Conference on Knowledge-Based Intelligent Information and Engineering*.

Llano, M. T.; Hepworth, R.; Colton, S.; Charnley, J.; and Gow, J. 2014. Automating fictional ideation using ConceptNet. In *Proceedings of the AISB Symposium on Computational Creativity*.

Llano, M. T.; Hepworth, R.; Colton, S.; Gow, J.; Charnley, J.; Lavrač, N.; Žnidaršič, M.; Perovšek, M.; Granroth-Wilding, M.; and Clark, S. 2014. Baseline Methods for Automated Fictional Ideation. In *Proceedings of the International Conference on Computational Creativity*.

Mihalcea, R., and Tarau, P. 2004. Textrank: Bringing order into texts. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Muggleton, S. 1991. Inductive Logic Programming. *New Generation Computing* 8(4).

Porter, M. 1980. An algorithm for suffix stripping. *Program* 14(3).

Romero, J., and Machado, P. 2007. *The Art of Artificial Evolution*. Springer.

Rumbaugh, J.; Jacobson, I.; and Booch, G. 2004. *The Unified Modeling Language Reference Manual*. Pearson Higher Education.