The FloWr Online Platform: Automated Programming and Computational Creativity as a Service

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

Abstract

We present recent developments in the Flowchart Writer (FloWr) project, where we have built a framework for implementing creative systems as flowcharts of processing nodes. We describe how the system has been migrated from a desktop application to a web portal and document the various features that the portal provides to support Computational Creativity research and development. This includes a node development package and automated chart development assistants. We detail how we have supplemented the online graphical platform with a web service API to enable developers to remotely access the features of FloWr through a programming language of their choice. This encompasses developing systems as flowcharts, together with running flowcharts remotely and also allows developers to publish flowcharts as web services. Importantly, the API allows Computational Creativity researchers to experiment with the automated development of creative software systems. To encourage this, we have also introduced simple models for automated software development into the FloWr API itself, providing a novel system for unsophisticated users to experiment with. We demonstrate the potential benefits of using FloWr, with case studies showing how the web portal has been used for both node and chart development by novice and expert users.

Introduction

In the FlowChart Writer (FloWr) project[1], we have built a platform for all Computational Creativity researchers to produce novel creative systems via GUI-based writing of *flowcharts* which pass information through processing *nodes*. The ultimate aim of the project is to form a community of users contributing to a corpus of nodes and flowcharts, which will enable the automatic generation of flowcharts, hence modelling creativity at the process level.

Since its introduction as a desktop application in (Charnley, Colton, and Llano 2014), there have been many developments with FloWr. One of the most significant changes has been the migration to an online version. FloWr users no longer have to download a huge Java desktop application, no updates are required and a variety of devices can be used to access the system. Nor are users restricted by the computing power of their device as processing is performed on our servers. Our main motivation for this migration, however, is a desire to provide a platform for interaction between researchers in the Computational Creativity community, by letting them share ideas, processes and resources, and collaborate on creative system development.

The main flowchart writing platform allows high-level creative systems to be developed. Users can also create new flowchart nodes to add novel functionality to the portal, increasing the power and scope of the systems that can be created. Our platform also provides Computational Creativity system design as a service through the FloWr API. This means that researchers are not restricted to the visual GUI and can access the full power of FloWr how they like, from whichever programming environment they choose. We are particularly interested in the possibilities for automated programming that this affords. Another new feature lets users expose flowcharts as standalone web services to allow other systems or users to access them. In the next section, we give details of the portal and highlight improved features of the online system over the previous desktop version. We follow this with two case studies highlighting the potential value of FloWr for novices and experts alike. We conclude with a discussion of future developments for the FloWr platform.

This work has similarities with the ConCreTeFlows project (Žnidarsic et al. 2016), where concept creation workflows have been implemented in a flowchart paradigm. That project uses Clowdflows[2] which, like FloWr, provides a portal for developing and sharing flowchart-based systems. Clowdflows was chiefly developed for algorithmic programming in machine learning and data-mining, with appropriate nodes. By contrast, FloWr has been developed for Computational Creativity collaboration and research and we are unaware of any other projects with these specific aims.

The FloWr Web Portal

In addition to improved facilities for writing flowcharts, users have access to a great deal of additional meta-level information describing what nodes and charts do. There is also an *Admin* area which gives access to numerous other enhancements, such as the *API* and *Node Development Package*. We describe the various aspects of the portal below.

[1]http://ccg.doc.gold.ac.uk/research/flowr/

[2]http://clowdflows.org/

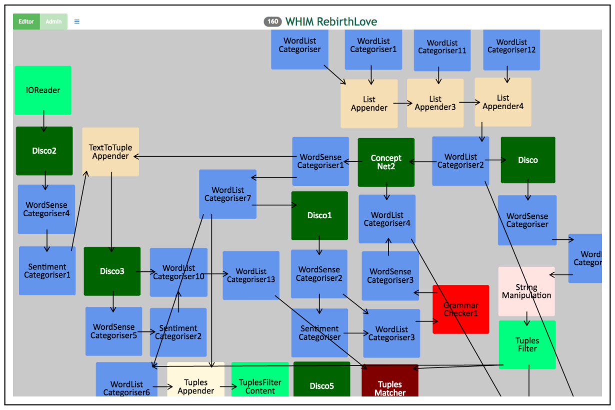


Figure 1: The online FloWr flowchart building interface.

Writing Flowcharts

The main interface, shown in Figure 1, is where users manually craft flowcharts. It is an improved version of the FloWr desktop application (Charnley, Colton, and Llano 2014). Flowcharts consist of nodes, representing self-contained processing elements, and arrows, which indicate how data is passed between nodes during processing. The core features of the desktop version have been retained including adding, removing, moving and re-sizing nodes, defining variables to highlight output, setting parameters, running charts and viewing output. Nodes are still written as stand-alone Java-wrapped code modules and the system still makes use of an underlying script syntax to describe the functional aspects of a chart that are independent from the graphical representation. General usability, feel and use-of-space has been improved by using modern front-end web development frameworks, as has cross-device/platform support. There are also improvements to GUI interaction, such as single-axis re-sizing and improved chart runtime feedback.

Node Information We have enhanced the information panel that appears on double-clicking a node, with additional information and the ability to open multiple instances. The *information panel* for a node shows the unique node id, its type, a bespoke label and a description of what the node does in the specific context of this chart, as well as tools to alter the node colours. The *input panel* shows, for each parameter, its name (with type tooltip), links to information about that parameter (see below) and a type-specific input control for setting the parameter. Static-value parameter setting has been enhanced to make it simpler and more fault-tolerant. Type-appropriate controls are displayed, such as checkboxes for Boolean values. Automated pre-validation ensures that, for example, numerical parameters are only passed numbers. Node developers can specify additional validation checks, such as maximum values, and bespoke data-types can be used to enforce regex-based validation. Developers can also choose which type of control, should be used for each input parameter, such as a textarea or textbox.

The *output panel* shows a tabbed list of all the defined output variables for the node (elements of its output that have been given specific labels). Output and variable definitions operate in the same manner as for the desktop version, using

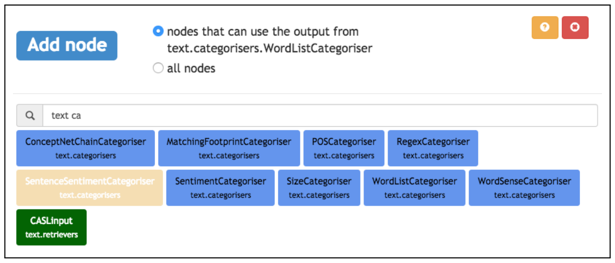

Add node
[✓ nodes that can use the output from text.categorisers.WordListCategoriser]
all nodes
text ca
ConceptNetChainCategoriser text.categorisers
MatchingFootprintCategoriser text.categorisers
POSCategoriser text.categorisers
RegexCategoriser text.categorisers
SentenceSentimentCategoriser text.categorisers
SentimentCategoriser text.categorisers
SizeCategoriser text.categorisers
WordListCategoriser text.categorisers
WordSenseCategoriser text.categorisers
CASLinput text.retrievers

Figure 2: The add-node wizard.

the same syntax. If the user has sufficient access rights – i.e. they own that chart – they can add, delete and amend variables, and we have made error notification clearer and more robust. Once run, any node output can be downloaded in JSON-encoded format. This is particularly useful for testing new nodes in the node development environment, as we describe below. The user can also inspect the output from the Java console, which is useful for debugging node errors and for applications where flowcharts are to be used from shell commands. As in the desktop version, the output of nodes can be locked so that outputs are cached which is useful for chart development and debugging.

Smart Assistance We have introduced wizards to help users create charts, as shown in Figure 2. For example, when the first node is added to a blank chart, the user will, initially, be shown only those nodes that have been tagged by the node developer as suitable for starting charts. Similarly, when a user wishes to create a link between two nodes, by holding down the control key to drag a new arrow between them, a wizard will suggest a subset of nodes which can, in some way, make use of the source node output. Similarly, when a user draws an arrow in to a node from empty space, or vice-versa, the new node wizard will consider which nodes might be appropriate to take/provide data from/to the target. These wizards use Java reflection to find potential data-type matches. Restricting nodes on this basis is very useful, given how many nodes are available, especially when dealing with certain artefact types, e.g., dragging an arrow out of an image retrieval node brings up a wizard showing only image-manipulation nodes. The wizards also provide a comprehensive node text search facilities to find particular nodes or domain-specific packages.

A new *MapHelper* wizard, shown in Figure 3, helps to create and manage data links between nodes, i.e., how data is passed between them at runtime. This appears whenever a new node has been chosen via the new-node wizards, above, or when an existing arrow is selected. This wizard shows the source, or output, node on the left and the target, or input, node on the right. Existing data maps are shown at the top of the dialog. To create new data maps, the user selects a parameter from the right-hand panel, whereupon FloWr uses Java reflection to review all the output of the source node to find any output or variables that match the data-type of that parameter. If the user clicks on one of these suggestions, the wizard establishes the data map, creating a new output variable where necessary.

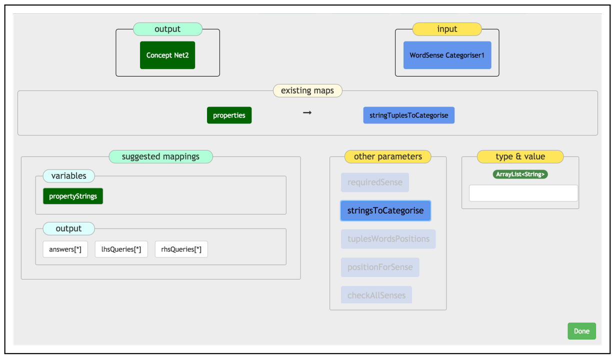


Figure 3: The map helper wizard.

Data Types The desktop version could already handle a large number of different data types and we have continued to expand this. For example, we have recently added images, which are represented internally as Java *Buffered-Images*. These can be large, depending upon the resolution, so thumbnails are used in output panels which give users the option to download the image at any resolution they choose, up to the full version. We have also introduced the notion of bespoke FloWr data-types. These are designed to enforce data-consistency by providing methods to support seamless front-end validation. For example, we have introduced a new *Float01* data-type which can only be instantiated with a string value that represents a valid float value between 0 and 1. The node provides a regular expression for validating front-end input strings accordingly. Node developers can use this to define bespoke data types and take assurance that values input to their node will be valid and consistent.

Menu The menu bar contains a number of new features. For instance, chart loading now includes lists per user and for recent charts. New *history* functionality takes regular snapshots of the chart and allows users to return their chart to a previous state. This also includes a facility whereby the user can take a named snapshot to better control versioning. There are other menu items for clearing the chart output, restarting the user-specific server (useful in debugging), highlighting run nodes and removing locks. The user can view the script underlying the chart and export the whole chart as an XML file. They can also view the supplementary information about the chart provided by the chart owner, as described below. Some of the charts that FloWr users have developed are quite large and contain many nodes. So, we have improved the way in which FloWr handles node positioning and introduced user-specific view profiles, which persist between sessions. The menu provides commands to re-centre the chart and help to find nodes that have been moved off-screen. Auto-layout using Graphviz (www.graphviz.org) has also been implemented.

Help and Information

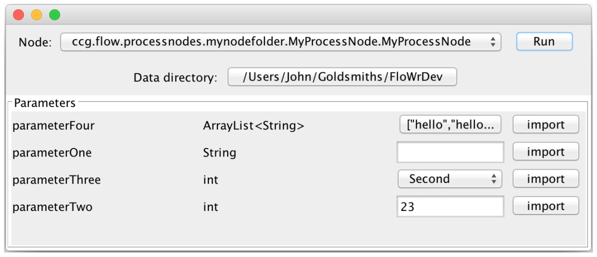
Every FloWr node or chart has a specific **owner** attribute, which is used to control access and editing rights. Only the owner of a chart can edit it and they can, optionally, lock it to prevent accidental changes. Charts can be *private*, visible to the owner alone, or *public*, where all other users can view and run, but not amend, that chart. To amend another person's chart, a user must take a copy and use that. Similarly, only node owners may download and make changes to a node's code and information (using the node development package described below). So, only node owners can download and upload new code or rollback versions. Chart Owners can provide an overview of the chart, bespoke node labels and context-specific descriptions for nodes using in-situ editors. Node Owners can provide additional information about nodes, which is available from various buttons next to node types and parameters. This includes an overview of the node, its default colours and whether it can be used to start charts, i.e. generates data from scratch (to inform the first-node wizard). It provides information about the parameters and allows specific input control, default value and validation options to be set. In particular, the node owner can specify drop-down options for a parameter, together with user-friendly replacement labels, if desired. This has replaced clunky source code constructs, which had a number of issues. Information can also be provided about the bespoke output objects, or sub-objects, that the node owner has created for their node.

Implementation

The portal uses a mixture of front-end web technologies, PHP, a relational SQL database and Java. To avoid cross-user data contamination, often caused by Java static variables, each user is given their own java server instance which handles their current chart. This also provides a load-balancing system. User-state is maintained server-side between sessions. Currently users must have a Google account to log in and their account must be unlocked using a code provided by the FloWr team. To maintain state consistency, users can only log in from one browser session at a time. We have taken steps to improve responsiveness by minimising client-server communication, e.g. by bundling calls in specific client use-cases. For speed of execution, chart runs are handled entirely by the back-end Java server with only minimal updates passed to the browser. Output for display is sent to the client piecemeal, with elements transferred only when viewed by the user. The system uses a mixture of sockets and file-system tools to transfer data around and minimise lag.

Admin Area

The admin area handles aspects of the portal that aren't concerned directly with flowchart writing. There are tabs for searching and managing charts, including moving them between the API and GUI or importing from XML. A tab for managing nodes, including downloading/uploading code, rolling back to previous versions and purging unwanted nodes entirely. Note that any changes to nodes must, currently, be carefully managed by node developers to ensure that existing charts aren't broken. The admin area also includes an ever-changing tutorial section. The developer section provides a link to download the developer package (see below). Other tabs in the admin area include recent news and developments. There is a place to provide feedback, a bug tracker for superusers and instructions for using the API.



Node: ccg.flow.processnodes.mynodefolder.MyProcessNode.MyProcessNode Run

Data directory: /Users/John/Goldsmiths/FloWrDev

Parameters

parameterFour	ArrayList<String>	["hello","hello...	import
parameterOne	String		import
parameterThree	int	Second	import
parameterTwo	int	23	import

Figure 4: The NodeTester application.

Node Development

We have created a developer package to help users create their own nodes. It includes the ProcessNode and ProcessOutput Java classes that underlie all nodes and their output (as described in (Charnley, Colton, and Llano 2014)). The package also provides a NodeTester application, shown in Figure 4, which lets developers parameterise a node, run it and inspect its output. NodeTester allows you to select a node to focus on, from those that you have in the local *processnodes* package. Below this is a DataDirectory selector. In the desktop version, this contained large libraries of static data, such as dictionaries or newspaper article archives. Previously users had to download these multi-gigabyte archives to their local machine if they wanted to use all the nodes. The online version keeps all this data on the server, so during development, node owners use a local DataDirectory and, when ready for release, an administrator installs the node's static data files on the server.

There is also a panel for setting parameters which uses a JSON format. Whilst not as user-friendly as the online GUI, it is as powerful and allows other data-types, such as lists-of-lists to be defined. Once parameter values have been set, they are stored locally as text. So, they can be edited manually and saved between development sessions. NodeTester allows users to import downloaded output from charts (see above) for use as values for parameters. Hence, developers can debug their nodes as though they were part of a larger flowchart, without having to continually upload and test their code changes.

Once a developer is happy with their node, they can upload it through the admin area of the portal. The meta-level node information that owners can provide about their nodes, as described above, is stored in XML format alongside the source code. If the owner desires, they can upload changes to this directly rather than using the online edit functions.

The FloWr Web API

One of the most exciting developments in the FloWr project is the API. This allows users to access the power of FloWr from within a programming language of their choosing. We describe this as Computational Creativity as a Service. Via the API, users can perform almost any of the actions that would be available to the online developer. As with the underlying script syntax, there is no need for notions of chart layout, colours, labels etc., and the API is predominantly functional rather than visual. So, for example, it is possible to set values for parameters but not, say, to re-position a node in some notion of a screen. Charts created under the API are kept separate from those created through the online GUI (this is changeable in the Admin Charts area). This is chiefly to remove the potential for portal to be swamped by a large number of automatically-created API charts.

The API is accessed by POST requests, which must specify the user's temporary 24-hour access token. As well as creating and editing charts, the API allows the caller to run them, as they would in the online GUI, and download the output they produce. Some of the commands available include all those for manipulating charts, parameters, variables, data maps etc. They also provide lists of available nodes, parameter information, output information and current chart state including run progress. Chart states are provided in JSON format, listing all the nodes and their parameters in a machine-readable format, similar to both the XML export and the underlying script syntax.

We are particularly interested in encouraging the API to be used for automated programming via automatic flowchart construction. To this end, the experiments in automatic flowchart writing presented in (Charnley, Colton, and Llano 2014) have now been run entirely separately to the core FloWr system, via the API. To foster such research, many of the API commands provide meta-data about the available nodes, their parameter types, output etc., which allows automatic programming approaches to make informed decisions about chart manipulation. In addition, the API has functions for users to upload and syntax-check flowchart scripts and XML charts, which provides another approach to automated programming that researchers could use.

Automation Features

In addition to providing API functionality to encourage automated programming, we have begun to add some automated programming features into the main FloWr GUI portal. In particular, users can call up the automated programming dialog box where they can re-run the simple flowchart construction experiments presented in (Charnley, Colton, and Llano 2014), where we asked FloWr to generate charts for creating poetic couplets from scratch. In the dialog box, the user can select the nodes to be placed into particular places in a template flowchart, sets of values to consider for particular parameters and they can specify a minimum level of output from the chart. They can then ask the system to generate a working flowchart. We are hoping that this demonstration of how FloWr might be used for automated programming could encourage researchers to perform their own experiments via the API, and we plan to greatly expand the online automation aspects.

Case Study 1: A novice user adds a new node

This case study describes the experience of a relatively new user to FloWr (author 4) who is familiar with the general features of the web API, and who wants to contribute to node development. His objective is to wrap one of the commands from the Microsoft Web Language Model API (www.projectoxford.ai/weblm) in a FloWr node, and use it in a sample flowchart. The command to be encapsulated is

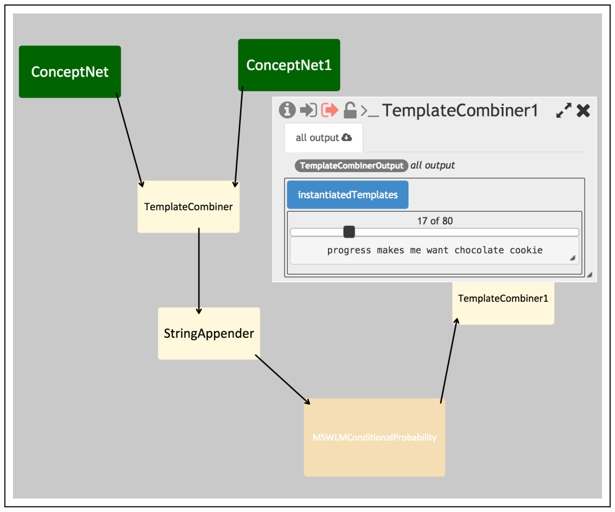


Figure 5: "Progress makes me want chocolate cookie": Simple interaction with a flowchart containing the new MSWLMConditionalProbability node

Conditional Probability that rates how likely it is that a particular word/phrase will follow a given sequence of words.

The user easily finds the `FloWrDev.zip` file inside the development area of FloWr. This contains the NodeTester toolkit, including `FloWrDev.jar`, a file of `INSTRUCTIONS`, and an outline of a sample node. The sample node does not have a lot of detail, so the user navigates to the *Nodes* tab and types "api" into the search box to track down code he had written earlier. There is a suitable existing node available as a template for modification, namely the node that wraps FloWr's own API. He downloads `FloWrAPI.zip`, which contains detailed and relevant sample code.

The sample files are renamed, placed in an appropriate local directory, then easily compiled and run under Java within the FloWrDev.jar environment. Java then displays a minimal panel for setting parameters and running the toolkit (Figure 4). After some further investigation, the user decides that he needs to track down another worked example that formats its output in a way that is more suited to the problem at hand. He returns to the web API and locates a node that is known to have output with suitable format, but this was created by another user so the *Get code* option is greyed out and unavailable. This time, he had another way to track down the code, but he submits a feature request asking for more public example nodes. The web UI has a *Feedback* where this sort of request can be made.

Adapting the FloWr API node to work with the Microsoft API, rather than the FloWr API itself, is straightforward. With the additional example in hand, formatting the node's output correctly is also easy. Following the strategy used in the example, a third file is added to support a supplementary class that organises output variables. During the adapting of the node, the toolkit gives access to standard Java debugging information and FloWr-specific runtime error messages.

The next step is to upload the code. On the *Nodes* tab in the web interface there is a button for this. Files are selected one by one using a file chooser. After some processing, the new node becomes available via the dropdown node chooser. Clicking on the *Description* button opens up an interface whereby documentation for the node and each of its input parameters and output variables can be added. Along with a plain-text description, this interface also displays the Java type information (which cannot be altered at this stage), and an interface that provides the ability to add or modify default settings and multiple drop-down options for the node. In this case, the user just fills in the plain text descriptions.

The node is now available for further experimentation. Although the NodeTester allows the user to try out various parameter settings for a single node, this is the first experiment with the node in context. Hints are provided (e.g., FloWr knows that a chart should typically begin with a node that retrieves text for further processing). It takes the user about half an hour to explore the available nodes and choose some that make a convincing demo (Figure 5). This flowchart uses a ConceptNet (Liu and Singh 2004) node to come up with food combinations using the template: [x HasProperty edible] + [x Any eat]. It concatenates these, and then uses the new node to decide which word combinations are likely to follow the word "eat". It then uses another ConceptNet node to select a putative cause: [x IsA change]. Finally, it combines the answers using a template combiner node: `b1Texts[*] makes me want b2Texts[*]`. Note that `b1Texts` and `b2Texts` here refer to the inputs to the template combiner. Along with *"progress makes me want chocolate cookie"* (Figure 5), other output with highly probable food items, as rated by by the new node, included *"grow makes me want cheese plate"* and *"become makes me want steak egg"*. Output ranked with low probability included *"become makes me want chocolate chinese_restaurant"* and *"progress makes me want dandelion goat"*. In the process of writing this flowchart, the user learns more about how the node-connection Wizard works: ctrl + mouse1 establishes connections between the nodes, and then the Wizard points out which available variables from a given upsteam node can be connected to a selected input parameter in the downstream node.

Along the way to this result, a few further ideas came to mind. Firstly, the node could be improved by allowing more input parameters, in order to make more expressive use of Microsoft's API . Secondly, additional text processing nodes could be created that quickly concatenate ArrayLists together to form n-grams for testing using the API (rather than using `TemplateCombiner` nodes). These ideas are easily addressed following the patterns described above. In total, the writing of the new node, deploying it within the online FloWr portal, then building, debugging and running a flowchart containing the new node took around 2 hours, which we believe is a reasonable time for a novice user to write a simple generative system. The new node contains 208 lines of code in total, of which approximately 40 are new. The experience was quite satisfactory to the user, who felt it was a good way for him as a novice Java programmer to get practice in the language.

Case study 2: An expert user's flowcharts

Here we present an account of the use of FloWr to develop a complex flowchart from the perspective of an expert user (author 3) who is not a developer of the core FloWr system (author 1). We also summarise this user's broader experience with the system. The flowchart we focus on is shown in Figure 1. This flowchart generates fictional ideas to be used, in this instance, in the context of generating original concepts for musical theatre pieces. Research using FloWr for fictional ideation has been carried out for around 2.5 years as described in (Llano et al. 2016). During this time, the expert user has added 41 new nodes to FloWr, which range over utility nodes, natural language processing, and domain specific nodes for fictional ideation and theory formation (Colton, Ramezani, and Llano 2014). An overview of nodes this user added to the system, and an example in each category, is presented in Table 1. A total of 17 flowcharts have been built to support fictional ideation, 9 of which have been released into production to be used in the European WHIM project (www.whim-project.eu) and 8 of them being more experimental. These flowcharts use an average of 35 nodes each, with the most complex flowchart composed of 73 nodes and the most simple one composed of 6 nodes.

Type	New	Example
Utility	11	RunShellScript: runs shell scripts
Natural Language Processing:		
Categorisers	2	POSCategoriser: annotates text with part-of-speech information
Combiners	5	ListAppender: appends two lists
Extractors	1	NamesExtractor: extracts proper names from text
Language	1	GrammarChecker: e.g. converts nouns from singular to plural
Manipulators	2	StringManipulation: changes text case
Matchers	2	TuplesMatcher: matches tuples in specified positions
Retrievers	9	WordNet: retrieves WordNet data
Theory formation	3	HR3: text to HR3 format (Colton, Ramezani, and Llano 2014).
Ideation	5	AudienceModel: ranks fictional ideas

Table 1: New nodes added by the expert user.

The starting point for our examination of the flowchart in Figure 1 is a set of templates that describe pre-defined, general scenarios that are to be completed by specifying either locations, attributes, or characters, etc., that form the foundation for the fictional ideas. These templates also form the building blocks of the flowchart and their structure guides the development effort. An instance of such a template is:

What if a PERSON_TYPE had to learn how to ACTIVITY in order to find true love?

The building blocks here are place-holders for the *person_type* and the *activity*. A ConceptNet node is used as the source of knowledge to retrieve a list of suitable concepts to fill the *person_type* place-holder. This is shown in Figure 6(a) block 1. As can be seen in this figure, various ConceptNet nodes are invoked, which retrieve ConceptNet facts of the form: [x, IsA, human], [x, IsA, occupation], [x, IsA, person] and [x, IsA, profession], where x is the person_type. Outliers are removed using *WordListCategoriser* nodes, and the results are appended to a common list through the *ListAppender* nodes. Building block 2 of Figure 6(a) retrieves facts of the form [x, Any, y], where y is then filtered to restrict only to verbs, through a *WordSenseCategoriser* node; the results can then be used in the *activity* place-holder. Finally, building block 3 combines these tuples into larger tuples of the form: [x_1, IsA, _, x_2, _, y], where $\mathrm{x}_1 \neq \mathrm{x}_2$. The template is finally filled in through the *TemplateCombiner* node, producing ideas such as:

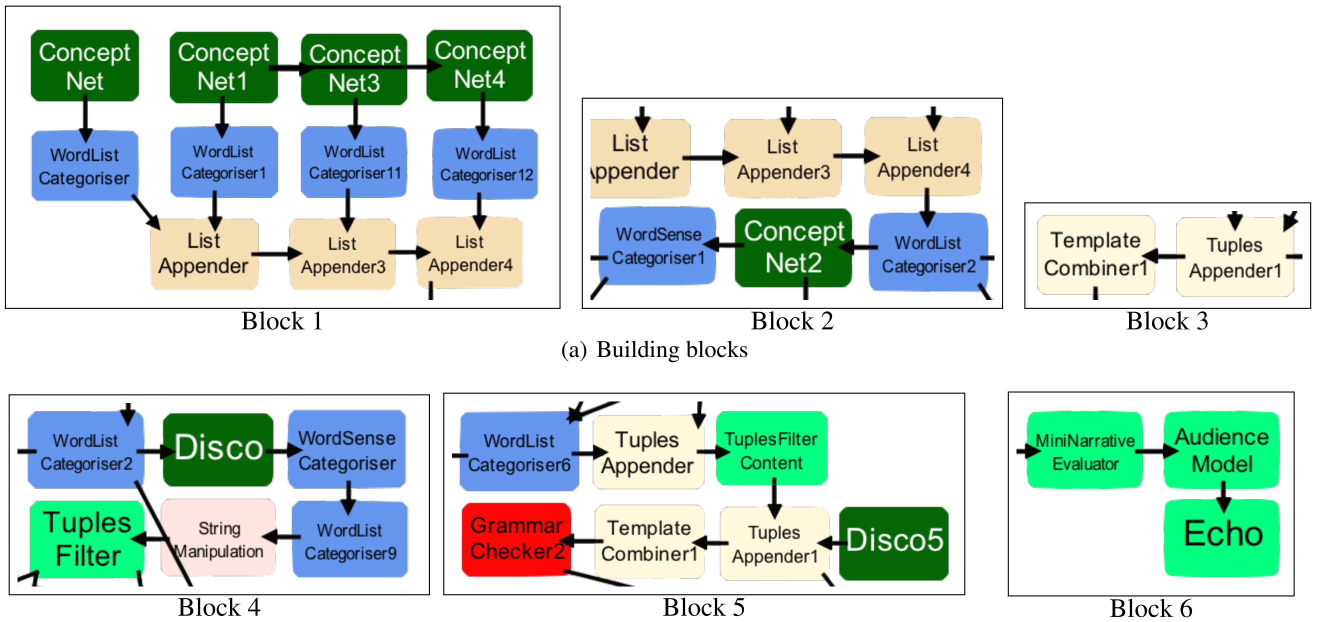
What if a banker had to learn how to fix a cat in order to find true love?

After a first version of a flowchart is finished, the output is analysed to decide if further work is required, which is usually the case. In the particular example followed here, two additional modifications were performed. These are shown in Figure 6(b). The nodes in blocks 4 and 5 retrieve representative qualifiers of the x_1 and x_2 concepts – for which there was not much data in ConceptNet. To achieve this, the *Disco* node, a linguistic tool (www.linguatools.de/disco/disco_en.html) that extracts related words using co-occurrences, was used. In particular, the 50 most common collocations for each concept were retrieved, and consequently filtered to keep only adjectives. Finally, block 6 in Figure 6(c) handles the evaluation of the ideas. This is achieved by connecting to an external web service that analyses their narrative potential through a set of measures; the results are subsequently fed into another external web service that contains an audience model that provides a ranking of the ideas. A possible expansion of the idea above, that is ready for this evaluation is:

What if a wealthy banker had to learn how to fix a cat in order to captivate an accomplished veterinarian?

Having access to different linguistic tools as nodes that support the retrieval and analysis of information has provided a useful framework for experimentation. In particular, being able to connect different tools (as illustrated above for the Disco and ConceptNet nodes) enables easy formation of a dynamic knowledge base to enrich generated output.

As can be seen from Figure 1, we have only explained a small subset of nodes used in the flowchart (we have focused on one of the possible templates). The flowchart deals with a total of 4 templates, each of which produces a different set of fictional ideas using common standard building blocks, contextual information blocks and evaluation blocks. This way of working provides a lot of flexibility. However, it also has the drawback that flowcharts can become cluttered with multiple nodes that could otherwise be encompassed in only one. For instance, if the ConceptNet node had either (a) an input parameter that takes a list of RHS or LHS queries in the form of text – it currently accepts these query parameters only in the form of an ArrayList of strings – or (b) if an alternative ConceptNet node existed with a text parameter, then all of the nodes in block 1 of Figure 6(a) could be replaced by a single node. Such functionality could be easily developed; however, following the line set out in alternative (a),



Block 1 Block 2 Block 3

(a) Building blocks

Block 4 Block 5 Block 6

(b) Context information. (c) Evaluation

Figure 6: Expert user processes in the fictional ideation flowchart, split into blocks.

nodes can become too complex, with an unmanageable variety of optional parameter combinations, and following the route set out in alternative (b), FloWr would end up with several ConceptNet nodes that only differ in the way the information is retrieved, so it could become difficult to track all of the variants. In any case, complex flowcharts are likely to be difficult to read because of the sheer number of nodes. A future improvement can be accomplished by having nodes that represent entire flowcharts, so for instance, block 1 in Figure 6(a) could be represented as a single node, which could be expanded at will to show the lower layers.

In summary, the experience with FloWr by the expert user has been very satisfactory in the sense of reuse and ease of experimentation. Challenges are present with all frameworks, and from experience of using FloWr in the context of fictional ideation, it is clear that the maintenance of large flowcharts (as in Figure 1) can be difficult, since the connections become confusing. Being able to use hierarchies of flowcharts would be facilitative. The format of data exchange is another challenge. As illustrated above with the ConceptNet node, the situation can be complex either for the developer or the user. We plan to implement a mechanism to handle the complexity of nodes in a flexible framework that can be tailored to different users with different priorities.

Conclusions and Future Work

We have described progress in the FloWr project, the long term aim of which is to study automating creativity at process level through the automatic construction of flowcharts for generative purposes. We have migrated FloWr to an online portal, with all the benefits that this affords, with an aim of creating a platform for interaction between Computational Creativity researchers. In addition, we have described new powerful additional features of the platform, such as the API, the node developer package and the ability to publish flowcharts as stand-alone web services. We are particularly interested in how the API will help and encourage researchers to experiment with automated programming. To illustrate the potential value of FloWr to Computational Creativity researchers, we presented two case studies. In the first, a novice user created a new node and a flowchart which used it in two hours, showing that the learning curve for the platform is not too steep. In case study 2, the accomplishments and experiences of an expert user showed that FloWr has real potential for building sophisticated creative systems.

FloWr has reached an important milestone in its development. The platform is now mature and performs all of the main functions for which it was built. We still foresee development work for the core system. However, at this point we have decided that, rather than second-guess which new features to introduce, we will focus our efforts on raising awareness of the system and supporting its adoption by new FloWr users by helping them to learn how to use various aspects of the system and contribute to the platform. We will then respond to feedback and direct our efforts accordingly.

Future feature improvements include better looping and conditional processing plus the introduction of new data-handling approaches such as tagging and a global data store during chart running. Although such enhancements should not be of major concern to API users, who can decide to run parts of the chart, and handle output, as they wish. Further improvements could include parameter validation using owner-defined regex checking or cross-parameter checks and better support for pausing and stopping chart runs.

Our automation experiments will continue. In particular, we are hoping that, as the corpus of nodes and flowcharts grows, we will have an opportunity to machine-learn regularities in the structure of flowcharts, and apply that knowledge to the generation of novel flowcharts.

In order to further support automated programming, it is likely to be important to expand on the bespoke data-type and supplementary meta-information, e.g. minimum values, that FloWr currently supports. Following the methodology of "Design by Contract" (Mitchell and McKim 2002), node authors would be able to make explicit statements of pre-conditions, post-conditions, and invariants. Sophisticated automatic programming clients could then reason about these specifications. There are a range of Java libraries that support this sort of annotation, e.g., the Java Modelling Language (Leavens, Baker, and Ruby 1998). Using such an approach, we plan to see whether an automatic programming tool could rediscover, for example, the patterns used in Figure 6(a)–6(c), and the way they fit together. This would be informed by work on reasoning about formal specifications. There is an established body of work in this area, much of it carried out in connection with theorem provers such as Coq (Bove and Capretta 2007; Tollitte, Delahaye, and Dubois 2012). Reasoning about program syntax and semantics is a recognised challenge "the pragmatic questions in this domain are far from settled" (Chlipala 2011, p. 340), and we hope to contribute to this field with reasoned automation over the FloWr system.

We intend to build on our initial work in encapsulating charts as nodes. We have developed a node which makes calls to the chart-run feature of the API and, so, we have been able to use entire flowcharts as single nodes in other charts. The next stage will be to resolve the issue of interface – i.e. how to re-describe specific parameters of the encapsulated chart as named parameters of the encapsulating node – so that traditional nodes and encapsulating nodes are indistinguishable. This is likely to require some development work in the core FloWr system, as the ability to pass in a list of encapsulated parameters with potentially differing data-types is not clear. Python and Clojure clients implementing the current API functions are available, and will be maintained and extended as the API evolves (https://github.com/holtzermann17/FloWrTester).

We want to enhance chart-sharing by introducing a system which, like view profiles, allows viewers of public charts to alter a subset of parameters, rather than having to take a copy. The subset of changeable parameters will be determined by the chart owner so that they can be changed to produce new chart output without breaking the chart. This will have some overlap with our work to encapsulate charts as nodes by introducing a notion of interface. In addition, we would like to enable users to switch between a number of pre-set chart parameterisations. We also plan to further extend the media types that FloWr can handle. The placeholder-viewer approach used for images is very effective and we expect to be able to easily expand FloWr to other multimedia domains, such as audio and 3D models.

The community features of FloWr will evolve. We will introduce sub-groups of users, context-based messaging, and comments and responses. We will supplement the basic public/private visibility system with the ability to open charts up to subgroups and introduce a similar system for node visibility and versioning. So, for instance, novice node developers will be able to ask for feedback from a select group before releasing their code to a wider audience. We hope that FloWr becomes a hub for Computational Creativity research, education and practice. Illustration with examples is very powerful and the ability to quickly set up a chart to demonstrate concepts with a natural language tool like Porterstemming (Porter 1980), could be highly effective in a range of scenarios. As per case study 1 above, we imagine a researcher who hears about a new piece of research or tool and is able to easily share it with the broader community by implementing it as a node and demonstrating its operation in a flowchart. Such illustrations have the power to be far more informative than, for example, writing up a research note or providing a link to a paper

Acknowledgments

This work has been supported by EPSRC Grant EP/J004049/1 (Computational Creativity Theory), and EC FP7 Grants 611560 (WHIM) & 611553 (COINVENT).

References

Bove, A., and Capretta, V. 2007. Computation by prophecy. In *Typed Lambda Calculi and Applications*. Springer.

Charnley, J.; Colton, S.; and Llano, M. T. 2014. The flowr framework: Automated flowchart construction, optimisation and alteration for creative systems. In *Proceedings of the International Conference on Computational Creativity*.

Chlipala, A. 2011. *Certified programming with dependent types*. MIT Press.

Colton, S.; Ramezani, R.; and Llano, M. T. 2014. The HR3 discovery system: Design decisions and implementation details. In *Proceedings of the AISB symposium on computational scientific discovery*.

Leavens, G.; Baker, A. L.; and Ruby, C. 1998. JML: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*.

Liu, H., and Singh, P. 2004. Commonsense reasoning in and over natural language. In *Proc. 8th Int. Conf on Knowledge-Based Intelligent Information and Engineering*.

Llano, M. T.; Colton, S.; Hepworth, R.; and Gow, J. 2016. Automated fictional ideation via knowledge base manipulation. *Cognitive Computation* 1–22.

Mitchell, R., and McKim, J. 2002. *Design by Contract, by Example*. Addison-Wesley.

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

Tollitte, P.-N.; Delahaye, D.; and Dubois, C. 2012. Producing certified functional code from inductive specifications. In *Certified Programs and Proofs*. Springer. 76–91.

Žnidarsic, M.; Cardoso, A.; Gervás, P.; Martins, P.; Hervás, R.; Oliveira Alves, A.; Gonçalo Oliveira, H.; Xiao, P.; Linkola, S.; Toivonen, H.; Kranjc, J.; and Lavrač, N. 2016. Computational creativity infrastructure for online software composition: A conceptual blending use case. In *International Conference on Computational Creativity*.