Augmenting Flow Diagrams Created by End-user Programs

Jonathan Lung
Department of Computer Science
University of Toronto
Toronto, Canada
Email: LungJ@cs.toronto.edu

Steve Easterbrook
Department of Computer Science
University of Toronto
Toronto, Canada
Email: sme@cs.toronto.edu

***Abstract*—Flow and causal loop diagrams can be used by content creators to illustrate how variables in a system impact one another. Such diagrams are used in educational settings to illustrate concepts like food cycles in an ecosystem and energy flows in climate models. We demonstrate how our tool, *Inflo-Graphic*, can produce interactive diagrams either by augmenting existing static diagrams produced using common tools users may already be familiar and have available such as the Gimp or Adobe Photoshop or by using a preexisting image or web page. Content consumers can use these interactive diagrams to visualize how changes to one part of a system can ripple through a system.**

I. Introduction

Complex systems are everywhere: we live in them (e.g., the earth, the solar system), we are part of them (e.g., economy, different communities), and we contain them (e.g., nervous system, digestive system). In order to understand them for ourselves or to explain them to others, it is sometimes helpful to draw diagrams. Flow diagrams and causal loop diagrams are useful for illustrating how different parts of a system impact and are related to each other. For example, a simple flow diagram might show how changing the temperature and/or pressure of a mixture in an equilibrium state would affect the concentration of various chemicals in the mixture. Another diagram might show how central bank interest rates, bond prices, and stock prices are all interrelated. An energy flow diagram of the earth might show how the energy from the sun is reflected, absorbed, and trapped by clouds, the atmosphere, and the earth's surface (fig. 1).

Flow and causal loop diagrams use arrows to show the direction of change propagation between components of a system; feedback loops that are present can be identified by cycles formed by the arrows. However, these diagrams do not convey how small (large) perturbations might have non-linear large (small) effects. That is, these diagrams do not always make it apparent what complex system dynamics arise from interactions between mechanisms [1].

One way of revealing these interactions is to run experiments on simulated models [3]. These experiments involve adjusting variables and watching the results. Experiments cannot be conducted on static diagrams as found in textbooks and standard lecture slides. In static diagrams with numerical values, values must be precomputed for a particular state.

"Guided rediscovery" with interactive models further improve understanding [3]. With guided rediscovery, educators provide a set of questions to learners with the expectation that learners will understand how various subsystems interact while attempting to come up with answers. For example, learners might be given the question "What will reduce the heating energy consumption of a home most?" Playing with a model could reveal how insulation efficiency, home size, desired indoor temperature, and weather affect energy consumption.

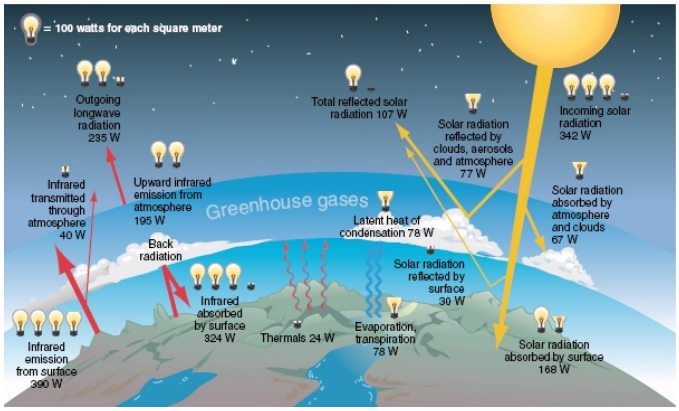

Fig. 1. An energy flow diagram showing earth's energy budget.[2]

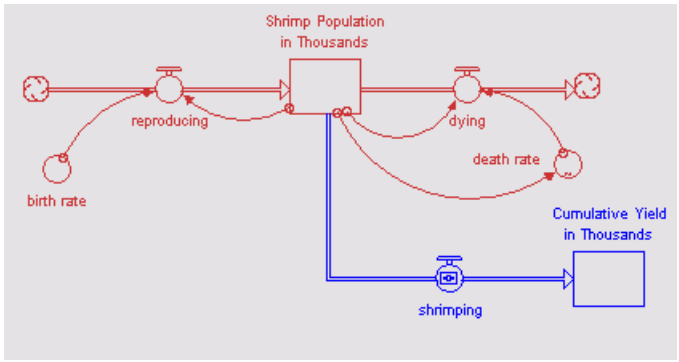

Fig. 2. An interactive stock-and-flow diagram produced by STELLA.[4]

There are many tools which can be used to create programmable/dynamic models. One way of thinking about a system is as set of stocks and flows. Existing stock-and-flow-based tools such as STELLA [4] and TRUE [5] can create these dynamic models; so can more general-purpose visual programming tools such as *Inflo* [6], [7] and LabVIEW. The models that are generated indicate which system components are related through lines and/or arrows. However, the visual

978-1-4673-0853-3/12/$31.00 ©2012 IEEE

result is utilitarian (e.g., see figure 2). Diagrams such as fig. 1 show there is room for aesthetic improvement. Elegant diagrams are not merely cosmetic. Learning is improved when things move from the abstract to the concrete visually appealing diagrams can help [8], [9].

II. Motivation & Goals

Considering the benefits of experimentation, guided rediscovery, and aesthetically pleasing diagrams, we wanted to empower people (especially in educational settings) to create elegant causal loop diagrams. We describe two approaches for creating visually elaborate interactive diagrams, with one approach built on top of the other: the *InfloGraphic* (*IG*) library, a JavaScript library that can be used to augment an existing static diagram by giving it dynamic capabilities, and *IG*UI, a user interface built on top of the *IG* library. Rather than build an entirely new tool for creating more visually elaborate diagrams, designers and developers can incorporate the *IG* library into their workflow (which we assume contains software such as the Gimp or Adobe Photoshop) and produce interactive diagrams. This library is not entirely accessible as it does require knowledge of Javascript. However, our second approach, *IG*UI, obviates the need to know JavaScript to create new interactive diagrams. We consider the lowering of barriers to creating visually pleasing diagrams one of the most important contributions of this work since it may increase use of flow diagrams.

III. Workflow with the *IG* library

A. Creating a diagram

Interactive *IG* diagrams begin life as static diagrams created by a *designer*. Any diagram that is stored in a compatible image format (e.g., JPEG or PNG), be it created in-house or from hiring a professional illustrator, is ready to be annotated to indicate regions of interest. Due to the wide range of sources, including a search on the Internet, the *developer* attempting to create an interactive diagram need not suffer for want of a static diagram.

B. Annotating the diagram

The designer or developer can open a suitable image in editing software such as the Gimp or Photoshop. These software packages permit users to divide an image into rectangular regions known as "slices". To create a slice, users drag a box around a part of an image using the slice tool; the enclosed region forms a new slice (see figure 3); *IG* repurposes each of these slices as a placeholder for a value from the dynamic model. These slices can be given a name, a fact which *IG* uses to its advantage. Using the image editing application, the sliced image can be exported as a set of images and an HTML *view file*. Although slice names are reflected differently in the output from these software packages, *IG* can automatically detect these.

C. Adding interactive elements to the diagram

The placeholder slices created during annotation (or, in fact, any image in an HTML file) can be overlaid with an interactive text box. Firstly, the developer needs to create an HTML *model file* to house the interactive model (in addition to the view file in section III-B). This model file must contain markup/code to perform the following steps:

Fig. 3. The "78" from the original static image in figure 1 has been erased and a slice named `CondensationHeat` put in its place.

1) import the *IG* library using the standard HTML `<script src>` tag,
2) specify the view file's file name,
3) add dynamic slices, and then
4) tell the library to run the diagram.

Aside from adding dynamic slices to the model file and specifying which file contains the view file, the rest of the model file is just boilerplate code.

To add dynamic slice, the developer needs to indicate

- the name of the slice in the view to be made dynamic,
- a default value for the dynamic slice, and
- a callback function.

```
ig.add_slice('mySlice', 12, function() {})
```

Each callback function returns a JavaScript hash table whose keys specify which dynamic slices are to be updated if this particular dynamic slice is changed and whose values are the newly calculated value. The callback function's arguments are the names of dynamic slices whose values need to be known in order to update the model. For example, to encode the relationship $velocity = \frac{distance}{time}$, the following anonymous function might be used for both the time and velocity dynamic slices:

```
function(time, distance) {
  return {
      velocity: distance / time
  }
}
```

In this case, if time or distance are changed by the user, the velocity dynamic slice's value will be updated according to the values for time and distance.

When all of the above steps have been performed, an interactive diagram has been created. In order to view it, the user opens the model file (which is just a standard HTML file) in his/her browser (see figure 4).

IV. Technical design

The *IG* library is designed to be a lightweight library to decouple the process of creating the artwork for a flow or causal loop diagram from the process of making it interactive. It allows users that have a webpage containing one or more images to overlay an interactive model when opened in a web browser. Because we wanted to facilitate the situation in which

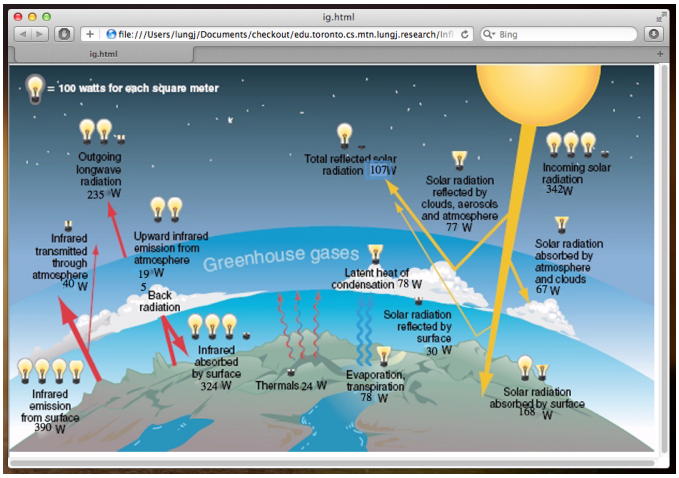

Fig. 4. An augmented energy flow diagram showing earth's energy budget with one of the text entry fields highlighted. As desired, it largely resembles the image in figure 1. The original values in the source image were erased and the diagram sliced, augmented, and populated with default values in under 10 minutes; however, no formulae were added – only default values.

the original diagram designer may not be involved with the augmentation of his/her diagram, the model-view-controller architecture was chosen because it provides a separation of concerns that aligned closely with the intended use cases. Within the MVC paradigm,

- **view** – the finished view is based around the flow or causal loop diagram created by a designer (sections III-A and III-B),
- **model** – the model is created by the developer (section III-C), and
- **controller** – the *IG* library serves as the controller.

Each of these components is stored in a separate file.

A. MVC – View (HTML, possibly with JavaScript)

Slices are usually used for the purpose of creating mouseover animations or optimizing an image for downloading. Because slices are simple to create, artists and non-artists alike can slice a completed image. *IG* uses these slices for a different purpose: the names of slices are used to couple the model to the view. To maximize compatibility not only with web pages created by image editors but also with HTML files generated by hand or other web editors, *IG* employs a variety of methods to match names used in the model and the HTML source. *IG* is designed to minimize interference with code in the HTML files that respond to mouse events, preserving rollover animations, form submit buttons, etc.

B. MVC – Model (HTML/JavaScript)

IG models are created in JavaScript and embedded in an HTML file so as to work in standard web browsers. The callback function design described in section III-C was chosen because it places no requirements on the programmer to adhere to any particular design principles. The actual model may be written using whatever techniques and technologies the programmer desires so long as they can interface with the browser's JavaScript engine. This could be as simple as providing a direct calculation formula in the callback function, as shown in the velocity example. However, the *IG* library is designed to accommodate coupling to any source including a JavaScript-based model or a model housed on a server. Since the full power of JavaScript is at the developer's disposal, dynamically generated images are possible.

C. MVC – Controller (JavaScript)

The *IG* library serves primarily as a controller. *IG* loads the HTML file containing the referenced view file and attempts to locate the dynamic slices in it that are referenced by the model. It then overlays these dynamic slices with pre-populated input fields for users to enter new values for the dynamic slices; these fields are, as expected, coupled to the model via the controller. To minimize interference with JavaScript code in the view, the controller runs the entire page in a sandbox and uses editable HTML DIV element overlays for user input/output: due to the way events are handled in web browsers, mouse movements and actions "bubble" allowing scripted actions attached to elements underneath the DIV overlays to fire.

The controller uses simple heuristics to determine the types of data in the model (to distinguish between strings, integers, and floats), performing automatic type conversions between the model and the view. It also uses software reflection to parse the callback functions defined in the model to simplify the API; the *IG* library automatically determines which dynamic slices' values need to be converted and passed to each callback function, eliminating the need for messy boilerplate code inside each callback function.

D. Split-file design

IG is designed to support the model and view being stored in separate files rather than the simpler single-file design. The reason for this is two-fold. The first has to do with streamlining the user's workflow based upon the expectation that many of diagrams will be sliced in the Gimp, Photoshop, etc. Instead of directly editing HTML file, these programs export HTML and supporting resource files, overwriting manual changes. Adding code for *IG* to these HTML files could become tedious; whenever a change to an image is made, all the *IG*-related code would need to be added again. Though this process could be partly automated, it would still involve an extra step to merge code. Instead, *IG* takes advantage of the run-time processing of the HTML view: the *IG* controller, when loaded in the HTML model file as part of the normal viewing process, loads the HTML view file and uses the web browser's parser to avoid dealing with variations in output between the software used to produce the view files.

The second major reason for the split-file design is to improve modularity. Doing an Internet search for "earth's energy budget" will yield many example diagrams that an educator could use for an interactive diagram. If an interactive diagram is produced with *IG* using a particular static diagram and, later, a nicer static diagram is found, it can be substituted after slicing the new image. The positions and sizes of the slices make no difference; the only thing that needs to be done to ensure compatibility is to keep slice names consistent between the old and the new HTML view files.

V. User interface

The *IG* library, a JavaScript file, by itself does not provide any convenient editing interface; developers may use whatever tools they wish. However, considering the intended user-base

includes educators that may not have experience with HTML, much less JavaScript, we prototyped a user interface for creating and editing *IG* diagrams. *IG*UI, a web application, unifies the processes of annotating (section III-B) and augmenting (section III-C) into one graphical tool. Though the purpose and uses of this tool are rather different from *Inflo*[6], a visual language based on directed acyclic graphs, we are reusing parts of the design language for expressing computation since people with no programming experience have found it usable.

The first step for producing a model using this service is to add an image or webpage to the user's library of static diagrams. This can be done by uploading an image or a web page as a ZIP file or entering a URL[1]. From here, instead of relying on an image editor (or code editor) to perform slicing, developers will be shown the items from their static diagram library in their web browser and can freely drag boxes to create slices that *IG* will track internally. This not only reduces the need to install an additional piece of software, it eliminates the need for an intermediate set of files to be managed by users. These slices can be named within the application.

Next, developers can create sets of models to connect to static diagrams. A static diagram may be augmented with one or more models. For example, in the earth's energy budget, one model might deal with incoming solar radiation while another might deal with infrared radiation originating from the earth. The two could be used to augment the same static diagram. There is a box reserved for entering equations into *IG*UI, much like a formula editor in a spreadsheet (except multiple *equations*, rather than a single *formula*, are entered). The "spreadsheet cells", however, are the slices defined by the user and may contain constraints. E.g., to disallow negative mass, the constraint `mass > 0` can be given where `mass` is the name of a slice.

Although systems may be cyclic in nature, when a user experiments by changing values in the simulation, we are interested in how changes propagate outward to reach a new equilibrium. If the new equilibrium can be computed by direct calculation (as opposed to be found iteratively via simulation), users do not need to do any complex programming tasks. As such, simple callback functions in *IG* may be entered using roughly the same interface components as used in *Inflo*. For example, if the dynamic slices `reflectivity`, `incident radiation`, `reflected`, and `absorbed` are defined, an *IG* callback for reflectivity can be specified as

```
reflected = reflectivity * incident radiation ,
absorbed = ( 1 - reflectivity ) * incident radiation
```

For complex models, programming is unavoidable; fortunately, *IG* allows developers to fall back on the JavaScript language where necessary. In all other cases, providing a dedicated editor that uses a formula editor along with a point-and-click interface rather than leaving developers to their own devices avoids the necessity for users to learn JavaScript, a set of APIs, and to wade through boilerplate code. When models are complete, developers will be able to share their interactive diagrams from within *IG*UI.

[1]In the current prototype, there is no diagram or model library (see below), preventing reuse. Background images and equations are selected/reentered for each dynamic diagram. Further, only a single equation can define a system at the moment. All other features work as described.

VI. Future work

*IG*UI will be evaluated with teachers when the fall school semester starts. In the meantime, we are seek approval for a study with students in a one-day art workshop for kids. In the workshop, we will give a "class" on energy efficiency in the home (for example, attic insulation, wall insulation, water boiler efficiency, and size of home) and have them create mixed-media art pieces of houses in small groups. Photographs of these art pieces will serve as *IG* diagram backdrops. Students can then slice these images identifying regions of interest (e.g., a box around the attic). After labelling these boxes by clicking on a slice and typing in a name like `size of home`, students can click on the equation editor at the top of the screen and enter an equation for the energy efficiency of the house. E.g., part of the formula may look like `size of home * heating energy per square foot * ( wall insulation efficiency * 70% + attic insulation efficiency * 30% )`.

An semi-structured interview with open-ended questions will be administered in an informal environment to assess understanding of concepts and reported enjoyment. Audio recordings throughout the day and screen capture will be used in tandem to augment the data.

VII. Conclusion

Flow and causal loop diagrams are useful for understanding complex systems and are a valuable educational tool. Elegant interactive versions of these diagrams provide additional benefits as a concrete vehicle for experimentation and guided rediscovery. Allowing educators and other individuals to augment new and existing flow and causal loop diagrams would allow these benefits to be realized by more people. The MVC model is appropriate for this situation; the *IG* library demonstrates why this architecture choice integrates well with different workflows. Thus, *IG* has made it possible to make existing diagrams interactive in ways which prior research has shown to be useful and can be used to rapidly prototype different styles of presenting models for research purposes.

References

[1] M. Cronin, C. Gonzalez, and J. Sterman, "Why don't well-educated adults understand accumulation? a challenge to researchers, educators, and citizens," in *Organizational Behavior and Human Decision Processes*, vol. 108, no. 1, 2009, pp. 116–130.

[2] (2007). [Online]. Available: http://cimss.ssec.wisc.edu/sage/meteorology [Accessed: 11 March, 2012]

[3] M. Schaffernicht, "Learning from rediscovering system dynamics models," *Systèmes dInformation et Management*, vol. 14, no. 4, pp. 87–105,114, 2009.

[4] (2012) STELLA modeling & simulation software. [Online]. Available: http://www.iseesystems.com/ [Accessed: 11 March, 2012]

[5] (2011, October) True-world: Temporal reasoning universal elaboration. [Online]. Available: http://www.true-world.com [Accessed: 11 March, 2012]

[6] J. Lung and S. Easterbrook, "Inflo: collaborative reasoning via open calculation graphs," in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, ser. CSCW '12. New York, NY, USA: ACM, 2012, pp. 1199–1202. [Online]. Available: http://doi.acm.org/10.1145/2145204.2145384

[7] ——, "A first look at end-user visual computation supporting sharing amp; reuse with inflo," in *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, sept. 2011, pp. 257 –258.

[8] P. L. Pirolli and J. R. Anderson, "The role of learning from examples in the acquisition of recursive programming skills," *Canadian Journal of Psychology*, vol. 39, no. 2, pp. 240–272, June 1985.

[9] G. Stenberg, "Conceptual and perceptual factors in the picture superiority effect," *European Journal of Cognitive Psychology*, vol. 18, no. 6, pp. 813–847, 2006.