Decision-Making Should Be More Like Programming

Christopher Fry and Henry Lieberman

MIT Media Lab
20 Ames St., Cambridge, MA 02139 USA
{cfry,lieber}@media.mit.edu

Abstract. *Justify* is an interactive "end-user development environment" for deliberation. Justify organizes discussions in a hierarchy of *points*, each expressing a single idea. Points have a rich ontology of types, such as *pro* or *con*, *mathematical,* or *aesthetic* arguments. "Programs" in this environment use inference rules to provide *assessments* that summarize groups of points. Interactive browsing modes serve as *visualizers* or *debuggers* for arguments.

1 Introduction

Online social media have given us a new opportunities to have large-scale discussions that help us understand and make decisions. But large-scale discussions can quickly get too complex. Who said what? Did anybody reply to a particularly devastating criticism? Is this redundant? Do the pros outweigh the cons?

Most people know basic concepts in decision-making, like weighing evidence, voting, or understanding dependencies. But an intuitive understanding is not enough to express ideas unambiguously, or when situations get complex.

We are proposing, essentially, an end-user development environment for online deliberation. Just like Eclipse is a development environment for Java, and Excel is a development environment for numerical constraints, we introduce the *Justify* system as an end-user development environment for rational arguments.

2 The Analogy between Deliberation and Programming

The analogy between deliberation and programming runs deep. Discussions are hierarchies of ideas. Programs are hierarchies of statements. In a discussion, people express reasons for believing or rejecting a single idea. Each of those reasons can, recursively, have reasons for accepting or rejecting it. Justify calls each idea, a *point*.

2.1 Points and Point Types

Since an argument is frequently a hierarchy, we adopt an outline view for the user interface. A point is shown as a single line in the outline, but it can be selected to see details or expanded to see subpoints.

Y. Dittrich et al. (Eds.): IS-EUD 2013, LNCS 7897, pp. 247–253, 2013.

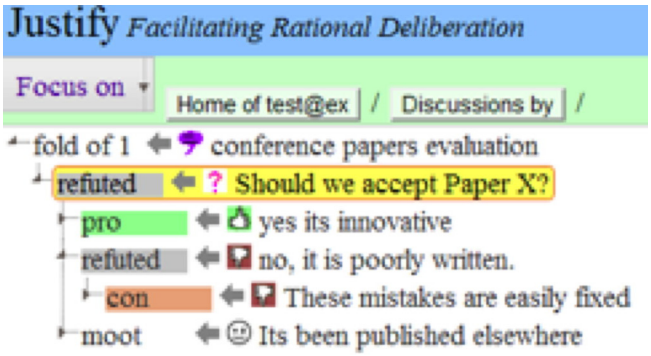

Fig. 1. An argument about whether or not to accept a conference paper

Programming has data types. Justify has *point types*. These are shown by icons that precede the one-line point description. Common point types are *question*, (question mark icon) which introduces an issue to be debated, and, for subpoints of a question, *pro* (thumbs-up icon) and *con*, (thumbs-down) for positions on the question.

Fig. 1 shows a Justify argument about the acceptance of a conference paper. The *question* is "Should we accept Paper X?". Below it are a *pro* point, "Yes, it's innovative", and a *con* point, "No, it is poorly written".

Below that appears another *con*, "The mistakes are easily fixed". Arguments are recursive. This refutes the criticism of poor writing directly above it. It is a *con* point because it is arguing against the criticism, so therefore it is an argument *in favor* of accepting the paper.

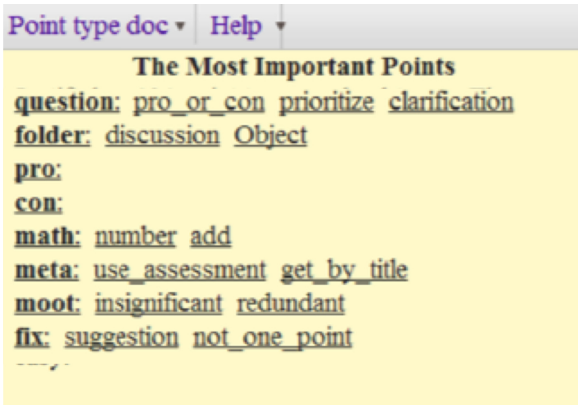

Fig. 2. Justify point types

2.2 Assessments

What does it mean to "evaluate" a discussion? An *assessment* is the result of evaluating a subtree of the discussion, and can be computed by arbitrary program code. Assessments for subpoints are like intermediate values in programming.

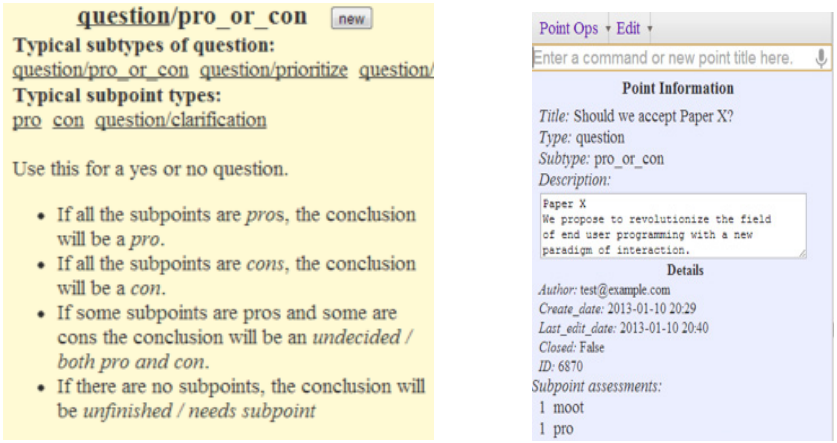

Fig. 3. Documentation on the question/pro_or_con point type, and a particular question's details

Assessments *summarize* their subtrees. A user can read an assessment and learn the result of the sub-arguments without reading through them.

Assessments appear to the left of the arrow on each line. Each point type has its own rules to compute its assessment. For example, an objection, with no subpoints, is assessed as *refuted*. So the "poorly written" criticism is refuted by the assertion that the "mistakes can be fixed".

The *moot* point type asserts that its superpoint is worthless, trumping any other assessment of that argument. Here we have a *moot* point, "It's been published elsewhere". Thus, the entire "Should we accept Paper X?" question is marked *refuted*.

2.3 Justify's Computational Model Is Like a Spreadsheet

The computational model of Justify is like a spreadsheet. Each Justify point is like a spreadsheet cell. The assessment, to the left of the arrow, is like the value of a cell. To the right of the arrow, the point type, represented by its icon, is like a spreadsheet formula that determines how the value of the cell is computed. The subpoints of a point, appearing below, are like the arguments to the computational rule that is represented by the point type. The point title is essentially a domain-specific comment.

For example, the math point type has subtypes that apply a given function to its subpoints; they can perform arithmetic makeing end-user programming in Justify like spreadsheet programming.

Like spreadsheets, Justify has a continuous computation model. When a point is changed, everything that depends on it is immediately recomputed. Assessments, which represent intermediate values, are always visible, facilitating debugging, as in the ZStep debugger [Lieberman and Fry 97].

Like other domain-specific programming languages, Justify presents a small set of primitives for common procedures. The design helps procedures "play nicely" with others so users can compose new capabilities on the fly.

2.4 Programming Concepts and Justify Concepts

Table 1. Programming concepts and Justify concepts

Programming concept	Analog in Justify	Programming concept	Analog in Justify
Program	Discussion	Returned value	Assessment
Source files	Justify shared repository	True/False	Pro/Con points
Built-in types/classes	Point types	List or array	Folder point
Subclasses	Point subtypes	Eval/run a program	Automatic (like spreadsheet)
Functions	Assessment rules	IDE	Browser on Justify web site
Object system	Prototype objects for points	Debugger	Expand/contract points
Expression in source code	Point in discussion hierarchy		View assessments

3 A More Substantial Example: A Program Committee Meeting

Let's return to the example about reviewing conference papers. Imagine that you are the Program Chair. The initial paper are completed. You would like to prepare for the Program Committee meeting.

Many conferences use prepackaged conference management software, such as EasyChair or Precision Conference. If the users follow the software's workflow, these work well. But with Justify, conference organizers can program their own.

3.1 Papers Reviewed by External Reviewers

Reviewers can use Justify to identify pro or con points about the paper, or assert a rating (on the conventional 1-5 scale).

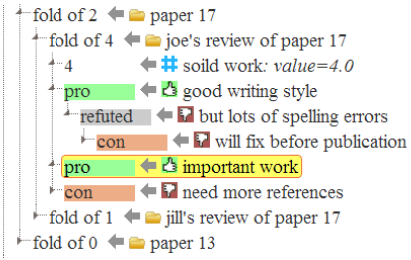

Fig. 4. Reviewers' discussion of Paper 17

3.2 Program Committee Discussion

Author rebuttal and reviewer discussion can be implemented as Justify points, as can the Program Committee discussion itself. Justify has access control via the *discussion* point type, allowing comments visible to the Program Committee only.

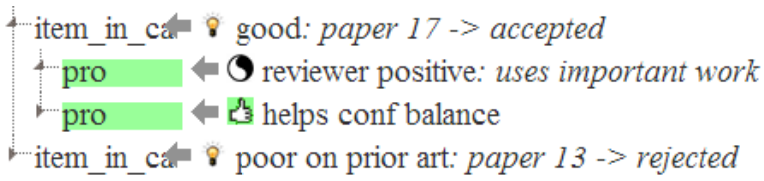

Fig. 5. Program Committee discussion. A PC member argues in favor, referencing a point made by one of the reviewers, who thought it uses "important work".

Rebuttals or PC discussions can target specific points of a review, packaging up the whole discussion for easy perusal by the Program Committee.

An author can rebut a reviewer point by creating a *use_assessment* point that references what the reviewer had to say in a different part of the hierarchy. In Fig. 5, the first *pro* point is a *use_assessment point* references the "important work" point.

3.3 Categories

Finally, the whole discussion is organized by using the *categorize* point type.

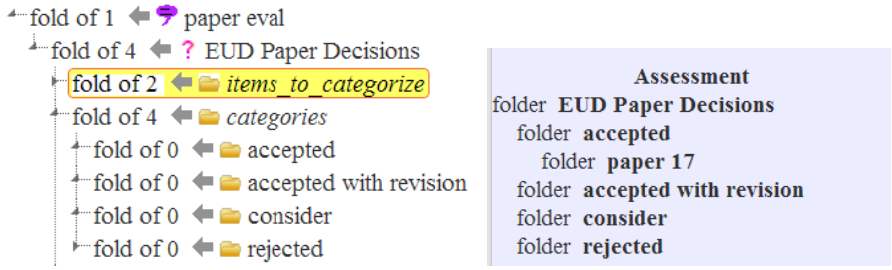

Fig. 6. Paper categories established by the Program Chair, and decisions

The Program Chair has set up four categories, accepted, accepted with revision, consider, and rejected. We might add other categories, for example, demote from long to short paper. The result is to put each paper in one of the four categories.

4 Usability Evaluation

We conducted a small usability study to answer: Did people understand the concept of Justify? What is its intended purpose? Would they use Justify ? We were worried that

the complexity of Justify's ontology of point types might limit usability. Although we only tested a few point types, results were positive. The point types, and hierarchical structure, did not prove a barrier to usability.

4.1 Experimental Method

Participants were shown a demonstration, then walked through two examples:

"Should I subscribe to a public bicycle sharing system? Should I purchase an iPad?

They then used Justify on whether or not to take a vacation in Hawaii.

4.2 Experimental Results

We tested 8 college students in their 20s. 88% said they understood the purpose of Justify (agree/strongly agree), 100% were confident in the basic operations on points, while 75% felt that way about using the more advanced point types. Respondents were split halfway about whether the ease of use was appropriate to the complexity of the example discussions, perhaps not surprising considering the example discussions were simple and Justify shines mainly in more complex discussions. 63% said they would be willing to use Justify for their own (presumably more complex) discussions. The one participant who strongly disagreed later clarified that her answer was due to the simplicity of the examples. Later work will test more complex scenarios.

5 Related Work

Argumentation systems have a long history, though we believe that this paper is the first to explicitly draw an analogy between argumentation and end-user programming. [Conklin, et al 2003] surveys landmark systems from Doug Engelbart's work on Augmentation and Hypertext from 1963 through NoteCards, gIBIS [Conklin 1988] and QuestMap through Compendium [Conklin 2003]. Conklin's work on Compendium incorporates the best ideas of the previous systems.

Compendium employs a 2-D graph of "icons on strings" showing links between nodes. This is semantically flexible, but requires more work in graphical arrangement and declaring link types than Justify's outline/hierarchy. We like Buckingham's work on Cohere and the conceptual framework described in [Buckingham Shum 2010].

We also like SIBYL [Lee 91] by Jintae Lee at the Center for Coordination Science directed by Thomas Malone. Fry worked in the early 1990's there. Malone's work of planet-wide importance continues at MIT's Center for Collective Intelligence.

Iyad Rahwan [Rahwan 11] tackles representing argumentation in the Semantic Web technologies of XML, RDF and OWL. This can standardize and share an ontology across the web, but pays little attention to the accessibility of the interface.

References

1. Buckingham Shum, S., De Liddo, A.: Collective intelligence for OER sustainability. In: OpenED 2010: Seventh Annual Open Education Conference, Barcelona, Spain, November 2-4 (2010)
2. Conklin, J., Selvin, A., Buckingham Shum, S., Sierhuis, M.: Facilitated Hypertext for Collective Sensemaking: 15 Years on from gIBIS. In: Weigand, H., Goldkuhl, G., de Moor, A. (eds.) Keynote Address, Proceedings LAP 2003: 8th International Working Conference on the Language-Action Perspective on Communication Modelling, Tilburg, The Netherlands, July 1-2 (2003), http://www.uvt.nl/lap2003
3. Conklin, J., Begeman, M.L.: gIBIS: a hypertext tool for exploratory policy discussion. In: Proceedings of the 1988 ACM Conference on Computer-Supported Cooperative Work (CSCW 1988), pp. 140–152. ACM, New York (1988)
4. Lee, J.: SIBYL: A qualitative decision management system. In: Winston, P.H., Shellard, S.A. (eds.) Artificial Intelligence at MIT Expanding Frontiers, pp. 104–133. MIT Press, Cambridge (1991)
5. Lieberman, H., Fry, C.: ZStep 95: A Reversible, Animated, Source Code Stepper. In: Stasko, J., Domingue, J., Brown, M., Price, B. (eds.) Software Visualization: Programming as a Multimedia Experience. MIT Press, Cambridge (1997)
6. Malone, T.W., Lai, K.Y., Fry, C.: Experiments with Oval: A radically tailorable tool for cooperative work. ACM Transactions on Information Systems 13(2), 177–205 (1995)
7. Mason, C., Johnson, R.: DATMS: A Framework for Assumption Based Reasoning. In: Distributed Artificial Intelligence, vol. 2. Morgan Kaufmann Publishers, Inc. (1989)
8. Malone, T.W., Klein, M.: Harnessing Collective Intelligence to Address Global Climate Change. Innovations 2(3), 15–26 (2007)
9. Minsky, M.: The Society of Mind. Simon & Schuster, New York (1988)
10. Rahwan, I., Banihashemi, B., Reed, C., Walton, D., Abdallah, S.: Representing and Classifying Arguments on the Semantic Web. The Knowledge Engineering Review 26(4), 487–511 (2011)
11. Speer, R., Havasi, C., Lieberman, H.: AnalogySpace: Reducing the Dimensionality of Commonsense Knowledge. In: Conference of the Assocation for the Advancement of Artificial Intelligence (AAAI 2008), Chicago (2008)