Using CogTool to Model Programming Tasks

Rachel Bellamy[1], Bonnie John[2], John Richards[1], John Thomas[1]

[1]Software Productivity Group
IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
rachel,ajtr,jcthomas@us.ibm.com

[2]Human-Computer Interaction institute
Carnegie Mellon University
500 Forbes Avenue
Pittsburgh, PA 15213
bej@cs.cmu.ed

ABSTRACT

In this paper, we describe the use of CogTool, a tool that enables non-psychologists to create cognitive models of user tasks from which reliable estimates of skilled user task times can be derived. We show how CogTool was used to compare a new parallel programming toolkit built on Eclipse, with Vim, a programming editor typically used in command line environments. This comparison was conducted to evaluate new parallel/scientific systems as part of the US Defense Advanced Research Projects Agency's High Productivity Computing Systems initiative. Our models indicate that for the four tasks analyzed, the new Eclipse tools are faster than the command line environments. Surprisingly, our models also reveal that despite programmers' preference for keyboard interaction in command line environments, mouse-based interaction is sometimes faster.

Categories and Subject Descriptors D.2 [**Software Engineering**]: Design Tools and Techniques, user interfaces.

General Terms Measurement, Design, Human Factors.

Keywords Programmer productivity, cognitive modeling, information foraging theory, CogTool.

1. INTRODUCTION

This paper describes one of the methodologies we are using to derive quantitative estimates of programmer productivity for the US Defense Advanced Research Projects Agency's High Productivity Computing Systems initiative. The scope and variability of work in parallel/scientific computing requires us to use multiple techniques. For relatively small codes, we are conducting experimental studies in which programmers are asked to parallelize a serial algorithm in a fully instrumented environment. For large multi-module codes, and for the porting of codes between systems, we are conducting retrospective interviews, extracting detailed use cases, and estimating time to completion. We are supplementing both of these approaches with modeling in order to be able to evaluate many tasks at a reasonable cost and schedule.

Our early modeling work built on the research of Brown and colleagues at IBM [2] who developed an approach to assess the complexity of computer-based tasks along the dimensions of number of steps, human memory load during the execution of the steps, and number of context shifts between sets of steps. The complexity model does not itself provide any time estimates. In order to provide these estimates in our earlier work, we made some simplifying assumptions based on average typing times for the command line interface case and Fitts' Law for the graphical user interface case, assuming roughly the same time per step in the two interface styles, a simplification known to be less accurate than other methods (see chapter 8 [3]). So while complexity modeling has allowed us to compare different tasks for overall complexity it has not allowed us to construct well-motivated estimates of the time it would take for the tasks to be completed – a requirement if we are to use the model results as part of our overall productivity gain computations.

2. COGTOOL

To better estimate task completion times, we have recently begun to use CogTool [7] a tool that allows UI designers to quickly and easily create valid Keystroke-Level Models (KLM) [3], and run them using the ACT-R cognitive architecture [1] to produce an estimate of task execution time for skilled users. Modeling based on these underlying formalisms has been successfully applied in a number of moderately complex real-world situations such as predicting behavior for telephone operators, IRS agents, and cell-phone use while driving. In a KLM, a task is modeled as a sequence of cognitive and motor operators. Each operator has a duration, based on prior empirical research, which estimates the average amount of time an experienced user would take to perform that operator.

CogTool is typically used to model routine computer tasks performed by experts. Programming, of course, is, in part, a creative, problem-solving task. So modeling here is

best reserved for routine parts of programming tasks such as code navigation, help lookup, and the like. Of course, if these routine tasks are cumbersome, imposing a significant interruption, they may have an adverse impact on the more creative aspects of the task. This interference of tool actions with task actions is known in the literature as the tool-task ratio [8]. When using tools, there are typically two problem spaces: the task space where task goals are addressed, e.g. find the line of code with a bug in it; and the tool space where tool goals are addressed, e.g. invoke find command repeatedly. The hypothesis is that when the ratio of tool operations to task operations is low then there is minimal task disruption. A good designer tries to drive the tool-task ratio towards zero and analyses provided by CogTool may contribute to that goal.

To create a model in CogTool, the user interface is first represented as a storyboard, with each state of the interface represented as a frame, each actionable interface item

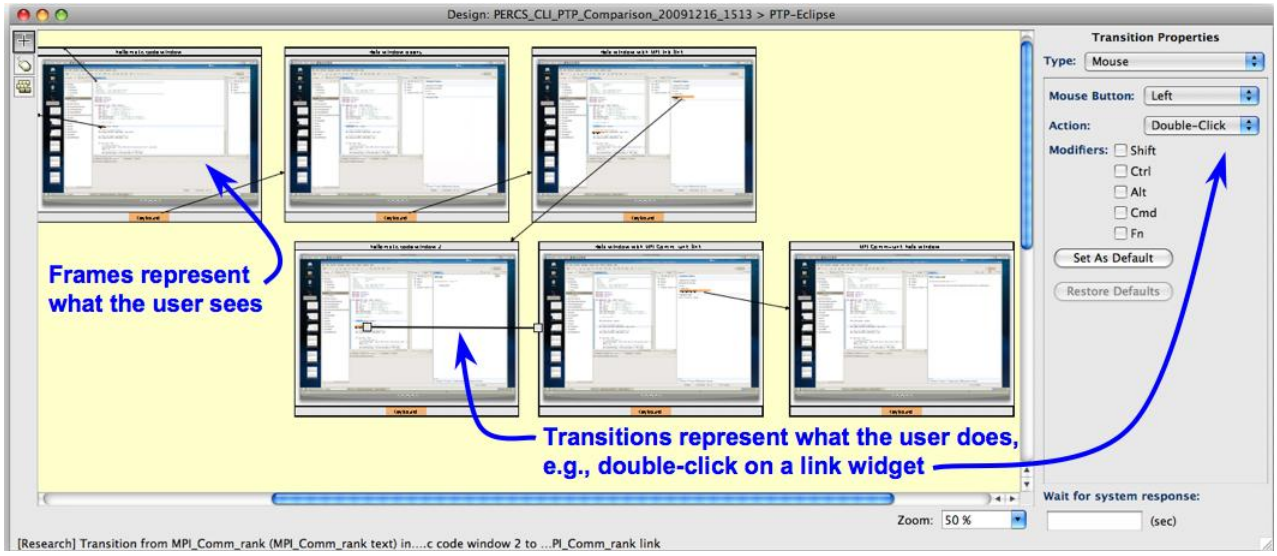

Figure 1 – A fragment of the PTP Eclipse storyboard for accessing help using the F1 key. The transition representing a double-click on a code object is selected (boxes at its ends), so its properties are shown in the properties pane on the right.

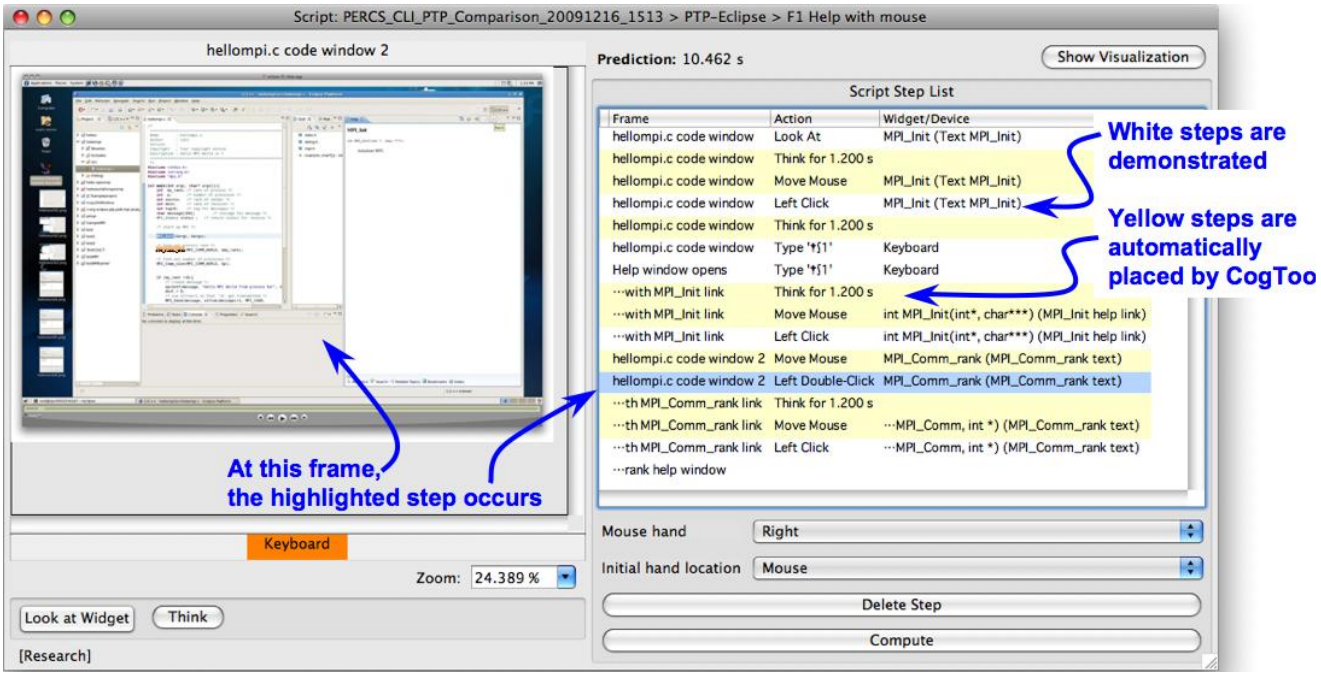

Figure 2 – CogTool script window. The frame on the left shows the Eclipse window. The next step is to double click on a piece of code in that window (highlighted in the list of operators on the right).

represented as a widget (e.g., link) or device (e.g., keyboard), and each action on a widget or device (e.g., mouse click, keys typed on the keyboard) represented as a transition between the frames. A piece of the storyboard for the PTP Eclipse environment is shown in Figure 1. These frames are from the storyboard for the task of accessing help using the F1 function key. Once the storyboard has been created, the analyst demonstrates the task on that storyboard by selecting a start frame, which appears in the script window (Figure 2), and performing the appropriate actions on that frame. CogTool records that action in the script step list (white steps on the right of Figure 2), automatically inserts additional operators based on prior research to create a valid KLM yellow steps in Figure 2), and shows the next frame. The analyst continues to demonstrate the steps in the task until it is complete and the entire KLM is built, as shown in Figure 2. When the analyst hits the "Compute" button, CogTool creates ACT-R code that implements the KLM and runs that code, producing a quantitative estimate of skilled execution time and a timeline visualization of what ACT-R was doing at each moment to produce that estimate.

Figure 3 shows the timeline visualization. The visualization window can be used to compare two demonstrations. The timelines are shown to the left, and on the right is the detailed trace of the automatically generated ACT-R model. The visualization is a time-line showing the use of ACT-R's eyes (purple), hands (red) and cognitive processes (gray). The visualization uses the well-known focus plus context UI design pattern, initially explored in the Fisheye work of Furnas [4]. The context is provided by the time-lines, shown at the top and bottom, which show the complete task. The timelines in the center focus in on the selected portion of the outer timelines. Clicking on an element in the focused time-line causes the relevant portion of the ACT-R trace on the side to be highlighted.

3. A COMPARISON OF 2002 AND 2010 PROGRAMMING ENVIRONMENTS

We used CogTool to compare times for a number of tasks done using the programming environment commonly used for parallel code development in 2002 – Unix command line and Vim editor – with the same tasks done using the latest parallel tool support built into the Eclipse Parallel Tools Platform (PTP). 2002 tools are the baseline comparison because we needed to compare productivity of tools commonly used in 2002 when the DARPA HPC initiative began with tools being created as part of the initiative with targeted completion in 2010. The tasks – creating a new program from a template, finding

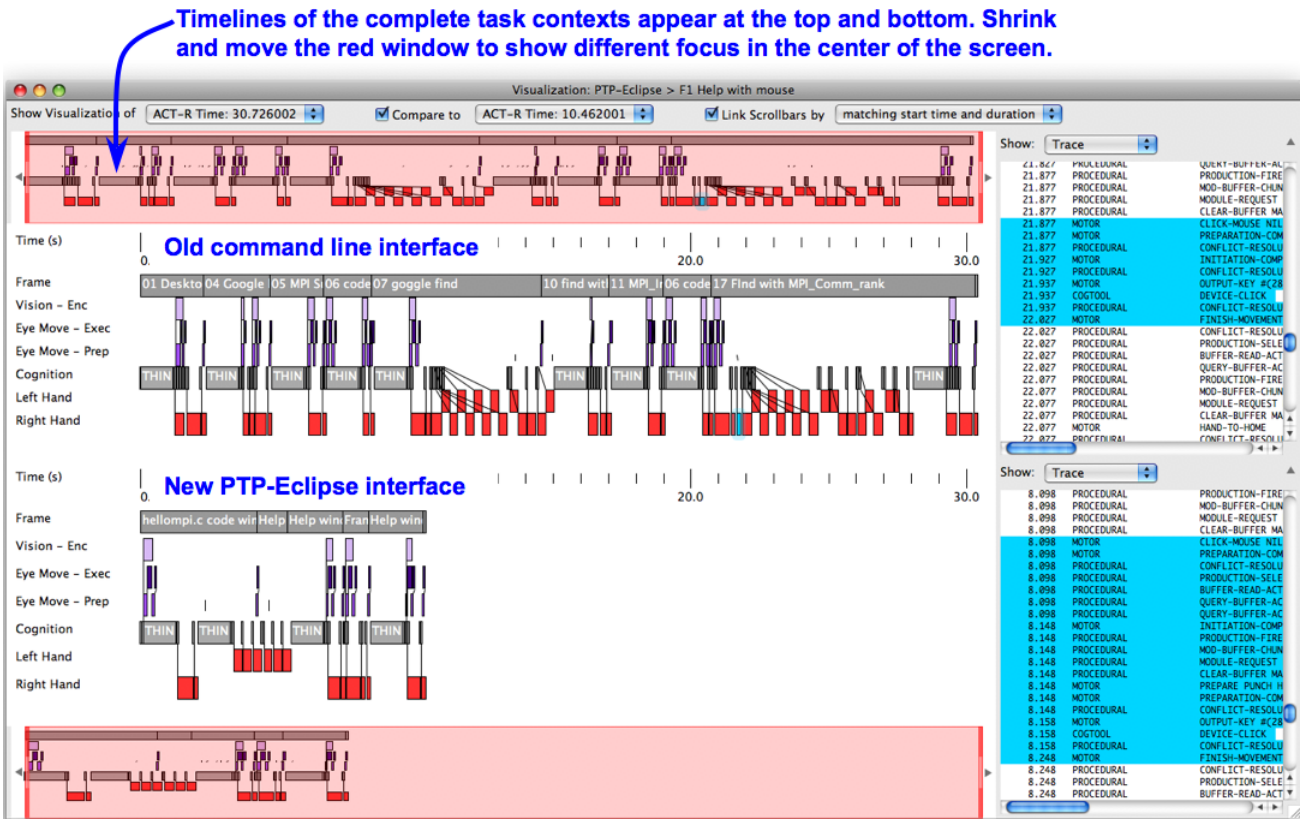

Figure 3 – CogTool visualization window.

documentation on an API, folding (hiding) sections of code to ease understanding, and finding barrier mismatches causing a deadlock – were chosen because they are representative of common parallel programming tasks that the designers of PTP aimed to make easier.

For each interface modeled we created a storyboard that contained the frames, widgets, and transitions required to do all the tasks, and then demonstrated the tasks on the storyboard. The results are displayed in a spreadsheet-like window (and indeed can be exported for use in spreadsheet applications), as shown in Figure 4. Here we can see that the new programming tools are faster for all tasks modeled so far. KLMs are known to provide estimates within 10-20% of what would be observed for empirical studies with skilled users [3, 7], so the differences shown in Figure 4 may be assumed to be reliable.

Looking behind the numeric predictions, Figure 4 shows that the overall time taken by an expert to do the F1 Help task using Eclipse PTP takes approximately one third the time of doing the task with the command line and Vim; the visualization shown in Figure 5 allows us to see how Eclipse PTP achieves this performance improvement. It takes about 5 seconds to open both help systems. In fact, it takes slightly less time to open the non-integrated browser-based help system via the command line. However with the integrated help system in PTP Eclipse, it is far easier to open help related to any particular code element. Examining the visualization reveals that, although both

* Project: PERCS_CLI_PTP_Comparison_20091216_1513

Tasks	Command Line	PTP-Eclipse
HellowWorld_mpi	Min: 114.405 s	Min: 40.090 s
HelloWorld_mpi with keyboard	161.111 s	
HelloWorld_mpi with mouse	114.405 s	40.090 s
F1Help	Min: 30.726 s	Min: 10.462 s
F1 Help with keyboard	30.726 s	
F1 Help with mouse	31.247 s	10.462 s
Code Folding	Min: 5.780 s	Min: 3.563 s
Code folding with keyboard	10.490 s	
Code folding with mouse	5.780 s	3.563 s
Barrier Analysis	40.149 s	11.833 s

Figure 4 – Results of modelling four tasks for a typical command line environment used in 2002, and for the new Eclipse PTP programming tools. (Some tasks were modelled two ways, resulting in the yellow task groups, discussed in Section 4.)

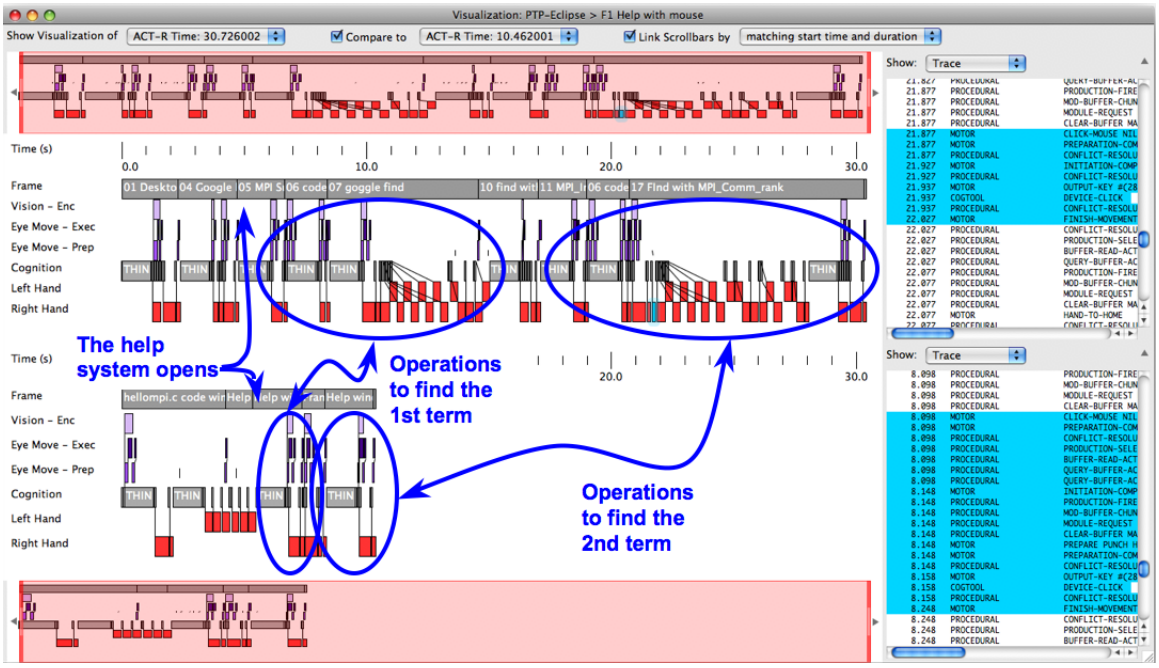

Figure 5 – Timeline visualization showing the key difference between the two systems.

systems require clicking to switch windows, the typing required to enter search terms into the command line's browser-based help significantly increases the overall task time.

4. REFLECTIONS ON USING COGTOOL TO COMPARE PROGRAMMING ENVIRONMENTS

We are pleased with the numeric predictions provided by CogTool, because they support our claim that the Eclipse PTP interface will improve the performance of skilled programmers over the 2002 command line interface. However, it is worthwhile reflecting on the experience of using CogTool for this endeavour to inform future efforts.

The hardest part of this CogTool modelling proved to be in understanding the tasks, especially given that parallel/scientific programming is a highly specialized skill and the fact that none of the people doing the analysis are actually using or designing the tools. Some of the specific issues we encountered were finding out which version of Vim was typically in use in 2002, finding out what features were supported in that version, and finding out idiosyncratic details of the Eclipse PTP user interface such as having to initially type function 'F1' twice, first to open the help panel and second to populate it. Defining the tasks required triangulating across multiple information sources such as old versions of manuals, and looking at old on-line discussions. Typically, CogTool is used in the early phases of development, by the designers themselves. These designers are familiar with the tasks they are trying to support and this problem would be greatly reduced.

As we defined the tasks to model we also made several detailed decisions relating to the fact that we were comparing two tools. For example, for the help task, we decided that we did not need to model reading the help material as this activity would plausibly be the same in both tool sets. Another set of decisions was related to ensuring that the tasks were equivalent across the tools being compared. For example, for the help task in the command line tool we decided that we should include the task steps of 'opening Google' and 'selecting the help pages from bookmarks' even though these steps were outside the tool being evaluated because these steps are analogous to the steps required to open the integrated help panels in Eclipse PTP.

When considering the Vim editor, we discovered that the version of Vim used in 2002 supported use of the mouse and copy and paste. However, we also knew from talking to programmers that many preferred to keep their hands on the keyboard as much as possible. This led us to decide to model tasks done using Vim twice, once with the keyboard, and again with the mouse (as shown in Figure 4), so we could compare the most efficient method to the new Eclipse PTP interface. We found that in the models, using the mouse makes no discernable difference in one task and two of the tasks considerably quicker. So why do experts prefer to keep their hands on the keyboard? Perhaps switching even between familiar devices causes a mildly disruptive context shift although we did not attempt to explain or model this personal preference here.

Another decision we made was to ignore system response time in our CogTool models even though CogTool models can include an estimate of system response time on any transition. It would have been extremely difficult to estimate realistic system response times for the 2002 interface, and time consuming to make these estimates even for the new PTP Eclipse interface. System response time is important for some usability evaluations because, for example, novice users can become confused by system delays with no feedback since they do not know whether they have done the right thing and then may start clicking again. However, we expect that skilled users will not be bothered as much by such delays since they know what to expect, and hence we felt it was appropriate to omit these delays.

Finally, we believe that making CogTool models compares favorably with running empirical studies to measure skilled performance time. Running empirical studies in this area is fraught with difficulties such as finding programmers expert in these interfaces (one programming editor is too old and the other one is too new!) and emulating realistic system response times from 2002. Furthermore, in the particular comparison discussed here, both versions of the tools had been implemented, but this is not always the case. In general it is beneficial to get usability data before months of implementation time has gone into making a working system.

Designer's of programming tools would benefit from using CogTool to test their designs prior to implementation. If a storyboard can be created, and a task demonstrated on that storyboard, then CogTool can be used to test the design before any code is written. Creation of models and interpretation leading to appropriate redesign does not require becoming an expert in KLM. When using CogTool, novice modelers can reliably create and interpret models [5, 6]

5. FUTURE WORK

Understanding the task times of expert programmers on tasks they are familiar with is, of course, valuable. But often programmers are doing unfamiliar tasks, or at least exploring unfamiliar source code. Extending modeling to cover these cases has required us to look, not just at the expert path through familiar tasks, but also the more circuitous and error-prone paths taken by programmers

when they are doing unfamiliar tasks. One promising way to model such task paths is based on the information foraging work of Pirolli and colleagues [10]. Information foraging theory is based on optimal foraging theory, a theory of how predators behave in the wild; predators sniff for prey, and follow scent to the patch where the prey is likely to be. Applying these notions to the domain of information technology, predators (people in need of information) sniff for prey (the information itself), and follow scent through cues in the environment to the information patch where the prey (needed information) seems likely to be. This theory has been found to be applicable to navigation when performing various tasks including debugging [9].

Information foraging has recently been implemented in CogTool (CogTool-Explorer), although applied so far only in web-search tasks [11]. Our work with this in parallel/scientific codes is just beginning but we believe it will allow us to extend our models further into the range of programming tasks encountered by parallel/scientific programmers.

6. ACKNOWLEDGMENTS

This work was supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. The third author's participation is supported by an Open Collaborative Research agreement with IBM. The views and conclusions in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA the U.S. Government, or IBM.

7. REFERENCES

[1] Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004) An integrated theory of the mind. *Psychological Review 111,* (4). 1036-1060.

[2] Brown, A. B. and Hellerstein, J. L. (2004) An Approach to Benchmarking Configuration Complexity *Proceedings of the 11th workshop on ACM SIGOPS European workshop*.

[3] Card, S. K., Moran, T. P., & Newell, A. (1983) *The Psychology of Human-Computer Interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates.

[4] Furnas, G. W. (1986) Generalized fisheye views. *SIGCHI Bulletin.* 17, 4, 16-23.

[5] John, B. E., (2010) Reducing the Variability between Novice Modelers: Results of a Tool for Human Performance Modeling Produced through Human-Centered Design. *Proceedings of the 19th Annual Conference on Behavior Representation in Modeling and Simulation* (BRIMS) (Charleston, SC, March 22-25, 2010).

[6] John, B. E., (submitted) Using Predictive Human Performance Models to Inspire and Support UI Design Recommendations. Manuscript submitted for publication.

[7] John, B., Prevas, K., Salvucci, D., & Koedinger, K. (2004) Predictive Human Performance Modeling Made Easy. *Proceedings of CHI, 2004* (Vienna, Austria, April 24-29, 2004) ACM, New York.

[8] Kirschenbaum, S. S., Gray, W. D., Ehret, B. D., & Miller, S. L. (1996) When using the tool interferes with doing the task. In M. J. Tauber (Ed.), *Conference companion of the ACM CHI'96 Conference Human Factors in Computing Systems* (pp. 203-204). New York: ACM Press.

[9] Lawrance, J., Bellamy, R., Burnett, M. (2007) Scents in programs: Does information foraging theory apply to program maintenance? *In Proceedings of Visual Languages and Human-Centric Computing*, IEEE.

[10] Pirolli, P. & Card, S. (1999) "Information foraging," *Psychology Review, vol. 106*, no. 4, pp. 643-675.

[11] Teo, L. & John, B. E. (2008) Towards predicting user interaction with CogTool-Explorer. *Proceedings of the Human Factors and Ergonomics Society 52nd Annual Meeting* (New York, NY, Sept 22-26, 2008).