Human-Centric Programming in the Large - Command Languages to Scalable Cyber Training

Prasun Dewan
Department of Computer Science
University of North Carolina
Chapel Hill, USA
dewan@cs.unc.edu

Blake Joyce
CyVerse
University of Arizona
Tucson, USA
bjoyce3@cyverse.org

Nirav Merchant
Data Science Institute
University of Arizona
Tucson, USA
nirav@email.arizona.edu

***Abstract*— Programming in the large allows composition of processes executing code written using programming in the small. Traditionally, systems supporting programming in the large have included interpreters of OS command languages, but today, with the emergence of collaborative "big data" science, these systems also include cyberinfrastructures, which allow computations to be carried out on remote machines in the "cloud". The rationale for these systems, even the traditional command interpreters, is human-centric computing, as they are designed to support quick, interactive development and execution of process workflows. Some cyberinfrastructures extend this human-centricity by also providing manipulation of visualizations of these workflows. To further increase the human-centricity of these systems, we have started a new project on cyber training – instruction in the use of command languages and visual components of cyberinfrastructures. Our objective is to provide scalable remote awareness of trainees' progress and difficulties, as well as collaborative and automatic resolution of their difficulties. Our current plan is to provide awareness based on a subway workflow metaphor, allow a trainer to collaborate with multiple trainees using a single instance of a command interpreter, and combine research in process and interaction workflows to support automatic help. These research directions can be considered an application of the general principle of integrating programming in the small and large**

Keywords—Cyberinfrastructure, workflow, awareness, recommender systems, visual programming

I. Introduction

By programming in the small, we mean creation of a program whose tasks are executed by a single operating system process, possibly interacting with one or more humans. Programming in the large is creation of a "program" or *process workflow* whose tasks are performed by multiple OS processes, again possibly interacting with one or more humans. Programming in the large, then, relies on programming in the small to create the code executed by the individual processes.

Such programming was first supported by the Unix command interpreter, called the shell. In fact, process composition is perhaps one of the most distinguishing features of Unix, supporting a philosophy in which each application or system program supports one function, and a multi-functional program is created by composing two or more unmodified existing programs. This principle has allowed operating system functionality to be implemented more concisely in Unix than in its predecessor, Multics. For example, a single "grep" program can be composed with an "ls" or "ps" process to search a directory and process listing, respectively, for a string. Such reuse has also been useful in application programming. For this reason, command languages in successors of Unix have all supported programming in the large.

II. Human-Centricity

Shell-based interactive command interpreters are sufficient but not necessary for programming in the large. It is possible to use, instead, Unix or some other API to programmatically connect processes together using a language (such as C) developed for programming in the small. Arguably, the purpose of command-interpreters is to support programming that is more human-centric – more interactive, collaborative, easier to learn, and/or easier to use for the task at hand. A similar argument can be made, using these characteristics of human-centricity, to argue that traditional command languages are more human-centric than traditional programming languages, whether the latter are used for programming in the small, or for programming in the large given a suitable API.

III. Large-Small Commonalities

The different degrees of human-centricity in the two programming granularities are both expected and surprising. If the two approaches were equivalent, then there would have been no need to support process composition in command languages. What makes the differences surprising is the argument that traditional programming and command languages are fundamentally the same, with the main difference being that they manipulate ephemeral (in-memory) data (e.g. scalars and arrays) and persistent data (e.g. files and directories), respectively. Heering and Klint [1] have in fact designed a monolingual environment that integrates traditional command, programming, and debugging languages. They have argued that even if such an environment is not practical, an integration exercise can enrich the individual languages. We refer to this principle as the granularity integration principle.

IV. Visual Programming in The Large

Both kinds of programming have evolved much since Heering and Klint's work – especially in increased human-centricity through visual programming. Visual programming in the small has, of course, received much attention in this conference. Figure 1 and 2 illustrate the use of the CyVerse cyberinfrastructure [2], originally called iPlant, to visually manipulate process workflows.

978-1-5386-4235-1/18/$31.00 ©2018 IEEE

Figure 1 demonstrates visual workflow composition. In Figure 1(a), the user creates a linear process workflow from the programs (FASTX) Trimmer, Clipper, and Quality Filter. In Figure 1(a), the user adds Quality Filter to the pipeline, not by typing its name, but by searching for it based on its name and attributes. Figure 1(b) shows the current programs in the pipeline, which can be edited by adding new programs, or by deleting or reordering existing programs. In Figure 1(c), the user connects the output of a previous program in the pipeline to the input of Quality Filter by choosing the output source from a menu that lists the potential options based on the preceding programs in the pipeline. This form of programming is akin to block-based programming in that in both cases, users can list, select and edit predefined templates.

Figure 2 demonstrates the subway model for visual workflow navigation, which, to the best of our knowledge, does not have a counterpart in programming in the small. In this model, programs in a predefined pipeline are visualized using a subway metaphor. Each predefined pipeline is mapped to a subway line and each program in the pipeline (e.g. Sequence Trimmer) is associated with a subway stop. Segments of the pipeline performing, together, some high-level task (e.g. Assemble Sequence) are put on separate branches. A user clicks on a stop to execute the associated program, and view and manipulate its output, before going to the next stop.

The three forms of programming (in the large) presented here, embody the general principle that a programming system can be made more human-centric, not only through more visualization, but also by making more decisions for the developer, that is, providing more restrictive, and hence easier to learn and use, specification mechanisms. Command languages are more flexible than visual workflow composition, which is, in turn, more flexible than visual workflow navigation. In terms of ease of use and learnability, the reverse order holds among these three programming abstractions.

V. Scalable Cyber-Learning

Ease of leaning, however, is still a major issue in all three forms of programming in the large. A command language is known to be difficult to learn and use. The visual alternatives, on the other hand, are not standard, and ever evolving. Thus, it is important to provide personalized and scalable training for cyberinfrastructure abstractions. These two requirements are apparently conflicting in that a there is a limit to the number of trainees a trainer can help. A further complication is that truly scalable training must, unlike the state of the art, be distributed.

We believe the granularity integration principle can be used to significantly improve this situation. Research on programming in the small has developed (a) awareness techniques for monitoring the programming of a relatively large number of novice programmers [3], and (b) automatic recommendation of solutions to novice programmers [4].

We are developing analogs of these techniques for cyberinfrastructures based on the following novel ideas: (1) *Distributed sticky notes*: Support a distributed analog of sticky notes [5] used in face-to-face instruction by trainees to indicate difficulties to trainers. (2) *Subway-based awareness*: When trainees are composing process workflows using command-languages or visual programming, create, for the trainers, a visualization of the trainee progress using the subway model, having each stop annotated with both summary and detailed information about the progress and difficulties of the trainees. (3) *Shell-based awareness*: Provide a trainer with shell commands to retrieve information about the trainees' progress, which can be more detailed than subway-based awareness, and can include, for instance, a representation of the history of operations executed by the trainees using the shell or its visual alternatives. (4) *Multi-user training shell*: Allow a trainer to collaborate with multiple trainees using a single instance of a command interpreter by injecting trainer commands into the command histories of trainees. (5) *Integration of process and interaction workflow*: Associate each process workflow to be created in a cyber training exercise with an interaction workflow – the kind used to constrain and define the work of employees in a business or government organization – and use this workflow to recommend next steps to those in difficulty.

CyVerse, being a production system, has an active training program, targeted at both domain scientists and students, that extends shell lessons provided by software carpentry [5]. Like software-carpentry, it requires face-to-face interaction with trainees. We propose to use our technical innovations to make this personalized training program distributed and more scalable, which will yield field data regarding their use. In addition, our planned evaluation includes controlled comparative lab studies

How these ideas may be fleshed out is a matter of research and is likely to benefit from conversations with conference attendees, who, in turn, would learn about the state of the art in visual programming in the large, its relationship to visual programming in the small, granularity integration, and our thoughts on using this principle to advance cyber training.

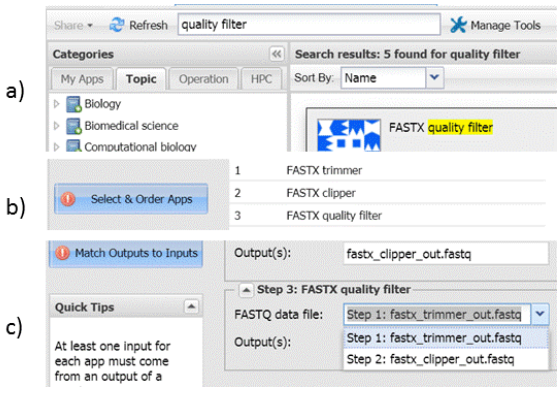

Fig. 1. Visually Creating a Workflow in Cyvese Discovery

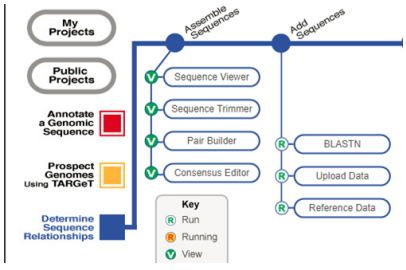

Fig. 2. Manipulating a Predefined Workflow in CyVerse DNA Subway

Funded in part by NSF grant OAC 1829752

REFERENCES

[1] Heering, J. and P. Klint, Towards Monolingual Programming Environments. ACM TOPLAS, 1985. 7(2).

[2] Merchant, N., E. Lyons, S. Goff, M. Vaughn, D. Ware, D. Micklos, and P. Antin, The iPlant collaborative: cyberinfrastructure for enabling data to discovery for the life sciences. PLoS biology ce, 2011. 14(1).

[3] Guo, P.J. Codeopticon: Real-Time, One-To-Many Human Tutoring for Computer Programming. in ACM Symposium on User Interface Software and Technology (UIST). 2015.

[4] Thomas W. Price, Y.D., Tiffany Barnes. Generating Data-driven Hints for Open-ended Programming. in EDM. 2016.

[5] Carpentry, S. Instructor Training. 2016; Available from: http://swcarpentry.github.io/instructor-training.