Towards Applying Complexity Metrics to Measure Programmer Productivity in High Performance Computing

Catalina Danis, John Thomas, John Richards, Jonathan Brezin, Cal Swart, Christine Halverson, Rachel Bellamy, Peter Malkin
IBM, TJ Watson Research Center
PO Box 704 Yorktown Heights, New York 10598 USA
1 914 784 7300
{danis, jcthomas, ajtr, brezin, cals, krys, rachel, malkin} @us.ibm.com

ABSTRACT

In this paper, we motivate and position a method for measuring the complexity of programming-related tasks. We describe this method, Complexity Metrics (CM), and give a brief worked example from the domain of High Performance Computing (HPC). We are using the method to help determine the productivity impact of new tools being developed for HPC by IBM. Although we argue that the CM method has certain virtues, we acknowledge that it is a work in progress. We discuss our strategy of complementing the CM method with knowledge we derive from applying other methods to better explore the complex issue of productivity. We end the paper with a discussion of some of the open issues associated with the CM method and plans for future work.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *complexity measures.*

General Terms

Measurement, Design, Human Factors.

Keywords

Complexity, metrics, measurement, productivity.

1. INTRODUCTION

The increased hardware performance of computing over the past few decades, and more recently the realization of highly parallel computing, have enabled the possibility of significant advances in a wide variety of application domains such as weather forecasting, modeling of complex physical objects, airplane design, biochemical engineering and data mining. Yet, that promise has not been fully realized largely owing to the difficulty of programming high performance computers which are typically highly parallel. It is well known that even serial computer programming is a very difficult and error-prone task. Some estimates are that between 50 and 75 % of overall effort and time in software engineering is devoted to the prevention, detection, and repair of errors in serial computer programs [13]. Dealing with highly parallel computing systems significantly increases the difficulty of prevention, detection and repair of errors.

The domain of HPC, often called scientific computing, is very broad both in terms of the kinds of codes developed and the users involved. Types of codes developed range from so-called "kleenex codes" which are typically developed by a single individual in a short period of time in order to try out an idea for an analysis and so are used only once, to multi-module codes developed over decades by large teams distributed in time and space. The large HPC codes are typically developed by multi-disciplinary teams that include both domain scientists, for example climate scientists knowledgeable about the atmosphere or about the behavior of clouds, and computational scientists, such as experts in data structures or in the communications characteristics of a particular HPC system.

In recognition of the perceived gap between the performance being delivered by parallel machines and the capabilities of programmers to harness the performance to solve scientific and business problems, DARPA sponsored the High Productivity Computing Systems (HPCS) program. This program which began in 2002, involved top vendors of HPC systems in a three-phase, eight-year effort to not only develop petascale machines but to also significantly improve the productivity of programmers who need to use such machines at scale. We are employed by IBM, one of two finalists in this program along with Cray, and are charged with measuring the productivity impact of the programmer tools that IBM is developing under the HPCS program. The program requires the vendors to demonstrate attainment of specific productivity enhancement targets attributable to the use of the vendor's tools (ranging from IDE's, to memory management tools to new programming languages) in order to satisfy program commitments.

Productivity is a complex construct to conceptualize, let alone to measure. Previous approaches have adapted constructs derived from economic theory to define productivity on an organizational level [e.g., 27]. Such approaches suffer from being highly abstract and fail to provide guidance as to how measurements for the various terms are to be produced. Since it is impossible to examine anything but the smallest of codes from start to finish, and since code development in HPC is skewed towards multi-module, long-lived codes, defining tasks for productivity assessment requires developing a strategy for sampling sub-tasks from the code development process for study. To this end, the HPCS community that organized around the DARPA program collectively defined "workflows" to specify the work of HPCS programmers and system administrators (we limit our focus here to code development scenarios). The workflows are high-level, idealized descriptions of the component tasks that comprise, for our current purpose, code development, including writing from scratch, debugging, optimizing and porting code. These workflows provide a shared landscape around which researchers can discuss the topic. We have undertaken observations of programming behavior in order to generate the greater detail that is necessary to study programmer productivity with our method.

Our approach, the CM modeling approach, is to define productivity for an idealized, expert individual in terms of performance in relatively small, individual tasks and to build up organizational productivity based on estimates of the frequencies with which the "measured" tasks occur in the work of programmers and weighted by the distribution of the skill level of programmers. The CM method highlights three aspects of task performance: the number of actions needed to complete a task, the number of data items the individual operates on while completing the task and the number of task context switches required in order to complete the task. These three dimensions are implicated in productivity because of their impact on the amount of time to complete a task and on the opportunity for errors they occasion. The argument is that task complexity is inversely related to programmer productivity.

The CM method we discuss in this paper is an analytic, rather than an empirical, approach. While the basic outlines of the method are broadly consistent with research in psychology related to task performance, we are engaging in targeted empirical investigations to better ground the method in empirical findings. Furthermore, the CM method is only one of a group of approaches we are using to try to quantify the productivity impact of our tools. As we discuss below, we are using these multiple methods to triangulate on our productivity assessments.

2. RELATED WORK

A number of related approaches have been advanced in both computer science and psychology. In general, the approaches stemming from work in computer science have attempted to measure productivity in the context of programming. These include Lines of Code (KLOCs), function points [1], cyclomatic complexity [22], software science [14], and COCOMO [4]. While each of these approaches captures some of the complexity of software, they were primarily motivated by concerns such as an attempt to compare programmers, to predict how long a project would take or to compare the inherent difficulty of different programs rather than to measure the difficulty of writing essentially the *same* program with or without various tools. It is this latter motivation which primarily underlies our own work.

In psychology, a variety of methods have been proposed to analyze and measure the complexity of cognitive tasks, of which programming serves as an example. These methods could theoretically be relevant to measuring the impact of tooling on programming. Although a full examination of these approaches is beyond the scope of this paper, it is useful to briefly review some of them in order to position our own approach. In order to understand these, it is useful to distinguish two related but separate threads of work. On the one hand, there are aspects of programming which often require creative problem solving and therefore methods arising from modeling of learning and problem solving are potentially relevant. On the other hand, programming also often requires activity that is fairly routine, at least for the experienced programmer, and here, approaches of modeling akin to the work of Card, Moran, and Newell [7] are potentially relevant.

Methods more appropriate to modeling the routine aspects of programming include, notably, GOMS [12, 16] and EPIC [19]. The human being, when viewed as an information processor, is obviously quite complex and applying such models to specific situations is typically time-consuming. In cases where an economically significant number of users will be using a product for a significant amount of time, such approaches can be quite cost-effective despite the up-front modeling costs [12]. For example, an analysis that identifies the potential to save a few seconds off of a process that is done by several thousands of workers can translate into significant savings. However, in other cases, such as HPCS where systems and applications are developed for a small number of users, an analysis of such precision is problematic. To address this issue, a tool, called CogTool [8] has recently been developed to make the construction of such models much easier. We are exploring the possibility of using CogTool to build models for routine programming tasks. However, in this paper we present an approach which is motivated by some of the same psychological research but is simplified to include only three major and relatively simple predictors of behavior: number of steps, context shifts, and memory load.

3. OUR OTHER APPROACHES TO MEASURING PRODUCTIVITY

3.1 Observations

As background for the more focused work of CM, we have carried out a series of "real-world" observations, interviews and surveys with HPC programmers and individuals in various "support" roles such as consultants and help desk personnel [9, 15]. Such work provides grounding for the definition of ecologically valid tasks for modeling with the CM method. It also provides data that will enable us to weight the tasks we will model based on their overall contribution to programmer productivity.

3.2 Interviews with tool makers

Because the suite of tools that is being developed in IBM to aid in the various tasks of parallel programming is extensive, we needed to identify the tools that would be expected to provide the highest productivity enhancement potential to programmers. To this end, we conducted a series of interviews with each of the tool developers. Based on these interviews, we then created a spreadsheet relating tools, availability, users, and workflows. This provides guidance to us for focusing our CM work where it is most appropriate.

3.3 Experiments

In addition, we have collaborated with various DARPA HPCS "mission partners" (i.e., labs in the United States that are potential users of the systems that will be produced through the program) to identify some representative HPC tasks. We have used somewhat simplified versions of these to gather empirical data about productivity, initially focusing on tools available at the start of the HPCS program in 2002 (hereafter, the 2002 baseline) and eventually comparing these to the tools that will be available at program completion in 2010. In addition to providing some direct empirical data about the overall productivity possible with the two sets of tools, these studies also provide further grounding for estimating the frequency of tasks as described above. We measure both time to completion and quality of code.

3.4 Use of Workstation Instrumentation

Our empirical work relies on the instrumentation of workstations and development environments for the automatic capture of programmer behavior at a fairly fine grain. In a previous empirical assessment of 27 novice programmers done in collaboration with colleagues at the Pittsburgh Supercomputing Center [10], we used a tool they developed called SUMS [24]. In our more recent work aimed at assessing the 2002 baseline, we have been using an open-source tool called Hackystat developed by Philip Johnson and his colleagues [17].

Ideally, productivity assessments would rely on direct measurements of programming in the "real world," documented through workstation instrumentation and supplemented with human observation. However, the large-scale nature of typical HPC codes makes this impossible. Even if one were to follow a code development from start to finish, over the years or decades required, the results would preclude generalization and thus be of limited use.

4. COMPLEXITY METRICS METHOD

The topic of psychological complexity, its definition and measurement, and its relationship to related concepts such as uncertainty, stress, and productivity is itself a wide-ranging and complex topic far beyond the scope of this paper (See Thomas & Richards [28] for a more thorough review). The complexity model we are focused on in this paper was originally based on the work of Brown, Keller and Hellerstein [6] and has been found useful despite the simplifications from full-blown psychological theory.

This model measures complexity along three dimensions: the number of steps in a process, the number of context shifts, and the working memory load (that derives from data operations) required at each step. It is capable of giving overall metrics of complexity for completing a given process with different tooling and is also capable of locating those particular steps that are particularly complex in terms of memory load. In this section we give a brief rationale for focusing on these three contributors to complexity.

4.1 Rationale for Number of Steps

One premise of the CM method is that the number of steps in a process can give an estimate of task complexity. We argue that the more steps that are required, the greater the complexity and chances of error for the programmer and consequently, the lower the productivity.

Of course, not all "steps" are equal and so using the sheer number of steps as a metric is somewhat limiting (we will expand on this limitation later). However, in most of the tasks we have studied so far (installation, configuration, simple administration and simple component programming and debugging tasks), the steps can be defined fairly objectively in terms of the task requirements within a given *style* (e.g., a graphical user interface (GUI) vs. a command line interface (CLI)). In GUI's, every new dialog panel or screen is considered one step. In line-oriented interfaces, every "Enter" is considered to mark the end of a step. These conventions presume some level of familiarity with the interfaces (which seems an appropriate assumption in the context of HPC). Typically, in comparing alternative products or various versions of one product, the "steps" are fairly similar in "size" (except as captured in the other two metrics; i.e., memory load and context shifts).

There are two additional dissatisfactions or shortcomings with the model as applied to straight-line processes. One is that it does not capture the complexity of the reading that is required either on the screen or with accompanying documentation in order to carry out a step. The second is that it does not measure how much background knowledge is required to decide which items need to be noted for future reference. (As expanded upon in Section 6, we are working to incorporate these nuances into the model). Nonetheless, in general, as processes gain more steps, there is a roughly monotonic increase in the chance of an error and certainly, an increase in time. As these tasks are performed in the real world, each additional step also increases the probability of being interrupted by some other task. Although the impact of interruptions is complex, they typically increase the chance of error and require the user to use some added time to recover state.

4.2 Rationale for Context Shifts

Context shifts were originally defined [6] in terms of computing contexts (e.g., server vs. client or operating system vs. data base). We have kept such changes as context shifts but broadened the definition to include shifts between applications or between installation components. The rationale is that if an installation requires the installation of three sub-components, these components often have somewhat different appearances and conventions.

Context shifts can be directly disruptive to working memory by requiring time and mental effort to orient to the new context. In addition, different contexts often employ different conventions and this can cause interference resulting in longer latencies, a greater chance of error, or both. For example, in some applications, clicking on the little red X in the upper right hand corner closes *that window* while in other applications, that same action may close *the application* (and *all* subsumed windows). In some applications, a SAVE command will utilize the user's current place in the hierarchy to determine where something will be saved, while other applications will not. Shuttling between these variations in conventions increases the chances of error. Even if no errors occur, the requirement to mentally "keep track" of which kind of application one is currently in probably impacts working memory load. Again, another chief advantage of using context shifts is that it is relatively easy to objectively obtain from the detailed task description of doing a task in a certain *style.*

4.3 Rationale for Working Memory Load

Working memory load [23] is concerned with the data used in a process and how it must be manipulated by the user. Working memory load is increased whenever the user sees something on a screen that must be remembered and used for some future step. Again, in detail, we know that the actual working memory load will depend on the type of item that needs to be stored and on the user's experience and strategies. However, as a first approximation, each new "item" that the user must attend to and remember increases felt complexity as well as increasing the chance for error. Even without error, it takes longer to recover a particular item from working memory if there are more items being retained. While this memory load is probably an important factor in task complexity of any sort, it is likely to be particularly disruptive in programming tasks which often require the programmer to mentally track an inordinate number of items. We should note that this "working memory load" is different both from the much more limited primary memory (for example, the ability to repeat a phone number that was just spoken by another person) and from what is stored in "long term memory" (for example, memory for well learned information).

4.4 Programming and Complexity Metrics

One of the main problems facing researchers interested in quantifying the productivity value of tools for the HPCS programmer (or for any programmer) is to define programming tasks that are both representative and tractable for measurement. This is true whether one is observing actual programmers or modeling their performance using techniques such as the one discussed here. In the portion of our work based on programmer observation we are focusing on the creation of fairly small "compact codes" proposed by the HPCS community as representative (the Scalable Synthetic Compact Applications, see [3]). In the work described here, we begin by modeling the more routine and repetitive aspects of code development (such as creating a new project, and looking up API documentation), gradually expanding our coverage to more creative parts of the programming task. We believe that the more creative parts of programming may benefit from another approach such as information foraging [21, 25].

5. BRIEF WORKED EXAMPLE

We illustrate the use of the CM method by analyzing the behavior required of a programmer seeking help on an MPI function he or she is incorporating into a program. In this example we contrast a hypothetical programmer Sam using the VI editor combined with a browser, with a hypothetical programmer Allie using the open-source ECLIPSE based PTP/PLDT IDE [11].

We begin the analysis with Sam looking at a VI editing session containing his partially written code. He has just typed the MPI function MPI_Reduce which he will use to collect the results from calculations done on many processors into a single value, but he does not recall the exact parameters the function utilizes. To get help he will go to a site on the web that has a tutorial on the MPI library and get the details he needs. The following are the actions he might produce in order to find the information and incorporate it into his program. Of course, different programmers will have different techniques to accomplish these same goals, for example going to manpages or books.

In this example, we assume that Sam already has a browser started on his workstation desktop. In Table 1, we see that his first action requires that he change context from his edit session to the browser session. This involves clicking on the browser icon (of the seven assumed to be) on his workstation task bar. Note that the number of alternatives among which he must choose is data dependent (discussed below under Open Issues) and will therefore vary by individual programmer. He then selects the "bookmarks" menu item from the menu bar, which is the fifth of seven items arranged on the Firefox browser menu bar. We then make another assumption, namely that Sam has the MPI help site bookmarked and thus does not need to search for a site that might provide appropriate help. He then scrolls down the list of bookmarks (the size again varies by individual) and clicks on the MPI help site (we have arbitrarily identified this as the 17th item in a list of 50[1]). He then has to find the MPI_Reduce function for which he needs help. He might do this in a number of ways. We assume that he types the function name into a search field. He then clicks on the link provided him and arrives at the appropriate help page. He reads the help page and must retain the information until he can enter it into his editing session. (Alternatively, he might use copy and paste to accomplish this goal.) After reading it, he returns to his edit session and in his final action, completes specifying the parameters for the MPI_Reduce function.

[1] Data details such as the number of alternatives for an entered data item do not currently impact the modeling results though they are tracked.

Table 1: Actions, context shifts, data items and memory load to complete help task using the VI editor and an MPI help site on the web.

ACTION	Context Shift?	DATA	MemoryLoad?
Change context to browser window by clicking on browser icon	✔	Select one of seven icons on task bar	✔ (name of function for which help is needed)
Select the "bookmarks" menu item		Select fifth item on browser menu	
Select the MPI help site		Click on 17th of 50 items	
Find the MPI_Reduce function		Type in MPI_Reduce and press enter key	(end of memory load retention period for function name)
Go to the MPI_Reduce page		Click on MPI_Reduce link	
(Read help on MPI_Reduce – not counted as action)			✔(name of 7 MPI_Reduce parameters)
Place focus on window with VI session	✔	Click on visible shell window	
(Enter parameters for MPI_Reduce – not counted as action)			(end of memory load retention period for parameters)

Table 2 shows the comparable actions for our hypothetical programmer Allie who is using the Eclipse PTP/PLDT IDE to do her programming. Like Sam, she has just typed the MPI function MPI_Reduce when she realizes she needs help to complete the specification. Her first action is to select the MPI_Reduce function name in her code. She then presses the F1 key on her keyboard to activate context-sensitive help. This opens a view in the IDE that lists the function name. At this point, in her third and final action, Allie clicks on the function name that is listed on the help view pane and the help content is made visible. When Allie returns to writing her code, placing her cursor on the MPI_Reduce function activates "hover-help" which lists the function name and its parameters, making it easy for her to enter the necessary parameters without having to commit them to memory or copy and paste them from the help materials.

Table 2: Actions, context shifts, data items and memory load to complete help task using the Eclipse PTP IDE.

ACTION	Context Switch?	DATA	Memory Load?
Select the MPI_Reduce function		MPI_Reduce string	
Invoke help through keyboard		Cntrl + Function 1	
(Read help on MPI_Reduce)			
(Enter parameters for MPI_Reduce)			

In the case of the PTP/PLDT tool, context-sensitive help allows the programmer to obtain additional information by merely selecting a string. The Eclipse PTP system uses the current context to automatically perform some of the necessary navigation for the user. In the comparable CLI case, the user must also explicitly remember what they need help on (or use the clipboard), then find the relevant help, and finally navigate back to the code (either remembering relevant information between steps, writing it down, or using more copy and paste steps). These additional steps and increased memory load will tend to increase time and errors. In this particular CLI case, we modeled going to a website to find the relevant information. We made the conservative assumption that the user had already bookmarked the correct file and had no trouble finding it or finding the correct function within the file. Even with this assumption, traversing back and forth to a browser window adds additional steps and increases memory load. In this case, the user had to retain seven parameters in memory; in actuality this memory load will vary by user behavior[2]. In this example, the user working in the CLI environment had to produce six actions, undergo two context shifts, produce six data items and retain two data items and retain data items in memory on two occasions. The comparable numbers for the IDE user are two actions, two data items and no context shifts nor any instances of memory load.

6. OPEN ISSUES

While we believe that CM is a promising approach for assessing the productivity of programmers using various HPC tools, we recognize that as a work in progress many open issues remain. In the remainder of the paper we discuss three that we are working on and discuss paths we are exploring to resolve them.

6.1 Levels and Types of Complexity

The problem of assessing the complexity of the tools our IBM colleagues are developing to support programmer productivity is made more difficult because multiple sources of complexity contribute to the task-tool-user nexus that we must model. We have identified four types of complexity that are present in any measurement and discuss here how each impacts the measurement task: *task complexity, tool style, tool implementation detail*, and *data complexity*.

The first source is *task complexity*. Task refers to the work the programmer has in mind when he or she sits down to work, for example developing a parallel implementation of the Smith-Waterman algorithm for local sequence alignment, adapting an existing program for modeling explosions to work with a new material or debugging code. While the range in possible task complexity is great, we hold this source of complexity constant in our comparison. That is, we look at the same task being done using two different tools, namely examples of tools used in 2002 and tools newly developed for the 2010 timeframe.

The tools we are testing contribute two sources of complexity. One source is the *tool style* or approach embodied in the tool, for example, a CLI versus a GUI. An example might be the use of a CLI editor such as VI or Emacs compared to an editor developed for embedding within an integrated development environment (IDE) like Eclipse. The complexity added by the tool style is the primary target of our measurement. However, it is further complicated by the presence of complexity due to the particular *tool implementation details*. This source of complexity has to do with how well the tool is designed rather than being an intrinsic property of the tool style. This is a type of complexity we have previously described as being "undue" or "gratuitous" as contrasted with "intrinsic" complexity [28] because it typically derives from poor design and implementation (Fred Brooks [5] referred to this as "accidental complexity."). Much of the gratuitous (or accidental) complexity is removed from tools through the iterative process of user feedback and tool re-design. In this regard, our measurements of the 2010 tools can be expected to be at a disadvantage compared to the baseline tools since many of the 2002 suite of tools will have matured through use and thus would be expected to have lowered gratuitous complexity compared to their initial releases.

A final issue in the application of the CM method derives from the *data conditions* under which the tool is tested. This is because the method takes into account the memory load involved in using a tool and one source of memory load is the programmer's operations on data. For example, generating a data item, such as the path where a file is saved, and then having to recover that path in a later step is said to place a memory load on the programmer. While this source of complexity, which is particular to the usage of the tool, can be held constant in the tool comparisons we do, it could contribute differentially to the overall complexity in many of the measurement situations we might encounter (e.g., like the opening of a file). In the example case, the user of the CLI environment might be generating a data item to specify the file from memory while the GUI user will be selecting it from a list of files. In either case, the programmer's task will be less complex when needed files are selected from a shallow hierarchy containing a small number of items compared with a deep hierarchy containing a large number of items. In general, we believe well-designed GUI's probably minimize the impact of the number of the data items relative to CLI environments but in our comparisons, we try to make comparisons as "apples to apples" as possible.

The indirect value that the CM tool provides for developers is primarily attributable to its usefulness in identifying the *gratuitous*

[2] As noted above, we could also have modeled the user's behavior to have copied and pasted the function parameters instead of retaining them in memory.

complexity that derives from poor design. Used as an analytic tool by the developers it provides the opportunity for the surfacing of usability problems (e.g., inconsistency in implementations of the "same" action in two different contexts) that have been built into the tool. In addition, it may help designers and developers become more conscious of how data conditions also contribute to the complexity of the user's task. In some cases this might be amenable to re-design (for example, rendering a selection as an auto-completion dialogue instead of as a pick list). We would expect the former to be less sensitive to increases in data complexity than the latter.

6.2 What unit of work equals an action?

In our discussion earlier, we noted that adopting various heuristics has enabled us to apply the CM method consistently because it is based on relatively simple definitions of what, for example, constitutes an action. Adopting the heuristics was a temporary, tactical decision made in order to allow us to apply the metric and thus gain experience with it.

A deeper conceptual issue related to what constitutes an action is illustrated with the following example from our experience in using the metric. Two of us (CD & JCT) were engaged in assessing our inter-rater reliability in applying the metric in preparation for training a third member of our team (PM) in its use. In the course of making some assessments, we first discussed various heuristics we wanted to apply including the aforementioned one menu choice equals one action. In the subsequent individual validation test measurements which included opening a file by traversing three levels of a menu hierarchy, a common action in programming tasks, one of us departed from the heuristic and coded it as a single action rather than three. In our subsequent debriefing he noted that by the time of coding, that portion of the task was quite familiar so he thought of it as a single action.

This raises the issue that expertise with using a tool or a style of tool has on the perceived complexity of the tool. Clearly, we found the perceived complexity of a sequence of actions decreased as we repeated that task component several times over the course of a short period of time. Thus, we realize that we need to include in our model a way for accounting for differential expertise with a tool. So, an experienced programmer might prefer to use keyboard shortcuts in a GUI and designers might be encouraged to provide them.

The previous example raises personal style as another contributor to the determination of what constitutes an action. Another member of our team (CS) is a very experienced programmer who repeatedly impresses his lab mates with his acts of computing prowess. Nonetheless, he traverses directory hierarchies one level at a time, frequently followed by an extra action to confirm his previous action (e.g., listing the contents of the directory he entered). He wants the feedback to avoid mistakes that could result in a great deal of work to correct (e.g., installing a multi-module system such as Eclipse in the wrong place in his directory structure). Consequently, individual differences in work style also impact the actual use of a tool and we need to consider how we might bring such a factor into our measurements. This factor can also interact with tool style, such that a tool which gives relatively little feedback following user action would likely lead to more defensive tool use.

6.3 Number of Steps: Is more necessarily bad?

In general, the interpretation of results from application of the CM is that a process that requires more steps to complete has greater complexity than one with fewer steps and therefore the one with fewer steps is "better" from the standpoint of productivity. This argument is built primarily on the increase in the chance of errors and opportunities for interruptions as well as on the additional time required to complete the process. The question to address here is whether more steps is necessarily bad.

An example will set the stage for this discussion. Imagine a programmer using an IDE to start a new project. In an Eclipse IDE, he would set several items, including the type of project he is starting, give the project a name and which compiler to use and its location. Now imagine two alternate, albeit somewhat extreme, implementations of the user interface. In the first case, all the fields for which the user has to specify a value appear on a single screen. In addition, the dependencies that exist between fields (e.g., deriving from Eclipse) are not indicated; instead the user has to infer them because seemingly at random, certain values for certain fields become unavailable for choosing. Now contrast this with a the second case, where each field is presented on a separate dialogue, but it comes largely pre-filled and simply requires that the user accept the default in moving across dialogues to complete the task.

In this example, it is not absolutely obvious that the second case, even though it has a potentially much larger number of actions is of greater *effective* (as opposed to measured) complexity than the first. True, the increased number of actions increases the opportunity for interruption and error, but since the data values are all or mostly all defaults, this task is very simple and may not increase the effective complexity for the user. In addition, the greater number of steps may not produce an increase in time since a single action that is mapped on to a multi-field dialogue with hidden dependencies may require just as much time or more than moving through a series of simple dialogues.

The larger number of steps may also not necessarily be bad from the standpoint of individual styles and the experience they embody. As noted above, some programmers have developed a cautious style, taking extra actions to confirm that a mistake has not been made based on their experience that some errors can be very costly to recover from. Interface styles might be differentially suited for a cautious style. One implication would be that applying the CM might require that we model a suite of users, including cautious ones. Furthermore, the need for providing state feedback has been discussed in the HCI literature.

7. FUTURE WORK

We intend to address open issues, of which the above three are important examples, by leveraging the work we are already doing in the empirical assessments of programming behavior and by also carrying out some additional targeted lab experiments. As we noted earlier, we are gathering data that allows us to examine programmer behavior as they code real world examples of problems. In addition to allowing us to assign weights that enable us to calculate the contribution of certain tasks to overall productivity, these observations will help us identify the alternative ways that programmers of different skill levels use to complete a task (e.g., remembering a set of parameters vs.

copying and pasting the information from one source to the target environment). It is very important for us to be able to ground the tasks we model in real world programmer behavior since there are multiple alternative ways to reach any goal.

We are also planning on using targeted empirical investigations to make further progress towards grounding the CM method in actual programming practices. This is particularly important in regards the question of what constitutes an action. One way that we plan to explore this is through lab studies in which we have programmers explain a task (such as starting a new project) to other programmers. In our studies we plan to make use of an established phenomenon in language studies in which a person explaining something to someone takes into account the knowledge level of their interlocutor. By varying the skill level of both the "teacher" and the "student" we should be able to discover whether expertise and other factors have an impact on the unit of behavior that constitutes an action.

The current model of behavior that is assumed by the CM method is quite simple and our empirical work is also serving to identify places where it must be extended. For example, in addition to considering skill level in the models, we also need to consider the complexity of data and the contribution of time to complete a task. We noted that the behavior models to which we apply the method presently consider data operations in the most rudimentary manner. We simply note if an item of data must be retained in memory for later utilization, but there are additional factors to consider. For example, as our brief worked example showed, there are tradeoffs between memory load and additional actions. We are further planning to explore how different types of data conditions (e.g., selecting among 2 or 50 alternatives, self or other-generated data items) and expertise in the task impact memory load.

And, finally, time is an obvious parameter that must be added to the model since the time to complete a task is a fundamental measure of how productive an individual is. One way we are considering bringing time into the model is to measure the impact of different tools on the completion of the more routine and repetitive tasks involved in programming on the completion of the more creative parts of the programming task, for example the coding of a complex procedure. The former more routine work is amenable to modeling and we are wondering if the complexity of completing such tasks might have a "carry-over effect" on the more creative aspects of programming which are not as generally amenable to our current method[3]. Does, for instance, the greater working memory load typically imposed by CLI style tools make it more difficult to imagine all the pathways of a complex algorithm? In closing, we believe that the CM method has promise and we are excited about the opportunities for expanding it and putting it on a sounder empirical basis.

7.1 The Iterative Development of the Tool

Although we argue that the CM approach is less time-consuming and requires less training than many alternatives such as building detailed user models or carrying out extensive user studies, the approach still needed to be supported with tooling. The original model we adopted took a detailed XML description of the task as input. We thought it unlikely that developers would use an analytic method that required this. Therefore, we developed a GUI tool to allow users to define the key model elements -- tasks, data, action steps, context switches, and memory load -- without having to directly write XML. The tool was used by a small group of people for some months. Interviews, observations, and spontaneous comments were all used to identify usability issues that were corrected in a later version. Later, a complete re-write was carried out based on further use on a number of different tasks by a number of different users. We will continue to iterate on the tool in order to minimize the effort required and to maximize the reliability of the method.

More broadly, we believe these same tools can have wide applicability within the software development process. For instance, since we are using a "simple" modeling approach, developers of any tool, system or application should be able to create a useful productivity estimate relatively quickly and easily (compared with either detailed modeling or extensive user testing in real contexts). Indeed, we have already used variations of this technique to provide feedback to developers to determine whether successive iterations of an installation procedure show increased or decreased complexity, to help locate sources of particularly complex interactions and to compare overall complexity among competing options.

[3] Selected parts of coding operations can be measured with the method, for example, where new functionality has been developed for operations such as barrier matching that is required for coordinating operations on multiple processors.

8. ACKNOLWEDGEMENTS

This work was in part supported by the DARPA contract HR0011-07-9-0002. We also thank two anonymous reviewers for their comments.

9. REFERENCES

[1] Albrecht, A. J. (1979) "Measuring application development productivity," in *Proc. Joint SHARE/GUIDE/ IBM Appl. Development Symp.*, Oct. 1979, pp. 83-92.

[2] Adamcyzyk, P. D. and Bailey, B. P. (2004). If not now, when? the effects of interruption on different moments within task execution. *Proceedings of the ACM conference on human factors in computing: CHI 2004*. New York, NY: ACM.

[3] Bader, D. A., Madduri, K., Gilbert, J. R., Shah, V., Kepner, J., Meuse, T., and Krishnamurthy, A. http://www.ctwatch.org/quarterly/articles/2006/11/designing-scalable-synthetic-compact-applications-for-benchmarking-high-productivity-computing-systems/. Website accessed February 21, 2008.

[4] Boehm, B. (2000). *Software cost estimates with COCOMO 2000*. Upper Saddle River, NJ: Princeton-Hall.

[5] Brooks, F.P. (1987). No Silver Bullet - essence and accident in software Engineering, *Computer* **20**, 4. 10-19.

[6] Brown, A. B., Keller, A. and Hellerstein, J. L. (2005), A model of configuration complexity and its application to a change management system. In Proceedings of the Ninth IFIP/IEEE International Symposium on Integrated Network Management. (IM 2005).

[7] Card, S.K., Moran, T., P. and Newell, A. (1983). *The psychology of human-computer interaction.* Hillsdale, NJ: Erlbaum.

[8] CogTool. http://www.cs.cmu.edu/~bej/cogtool/.

[9] Danis, C. 2006. Forms of collaboration in high performance computing: Exploring implications for learning. Proceedings of the conference on Computer Supported Cooperative Work, Banff, CA.

[10] Danis, C. and Halverson, C. The Value Derived from the Observational Component in an Integrated Methodology for the Study of HPC Programmer Productivity. In The Third Workshop on Productivity and Performance in High-End Computing (P-PHEC-3), Austin, Texas, 2006.

[11] http://www.eclipse.org/proposals/eclipse-ptp/.

[12] Gray, W., John, B., Stuart, R., Lawrence, D. and Atwood, M. (1990). GOMS meets the phone company: Analytical modeling applied to real-world problems. *Proceedings of IFIP Interact '90: Human Computer Interaction.* 29-34, Cambridge, UK.

[13] Hailpern, B. and Santhanam, P. (2002), Software debugging, testing, and verification. *IBM Systems Journal,* **41**(1), 4-12.

[14] Halstead, M. (1977). *Elements of software science.* New York: Elsevier.

[15] Halverson, C. Unpublished field notes, 2006.

[16] John, B. E. and Kieras, D. E. (1996). Using GOMS for user interaction design and evaluation: which technique? *ACM Transactions on Computer Human Interactions* **3** (4), 287-319.

[17] Johnson, P. Hongbing, K., Agustin, J., Chan, C., Miglani,J., Shenyan, Z. and Doane, W.E.J.. (2003), Beyond the personal software process: Metrics collection and analysis for the differently disciplined, In *Proceedings of the 25th International Confefrence on Software Engineering,* 641-646.

[18] Kepner, J. HPC Productivity: An Overarching View. (2004). International Journal of High Performance Computing Applications, 18, (4), 393-397.

[19] Kieras, D. & Myers, D. (1997). An overview of the EPIC architecture for cognition and performance with applications to human computer interaction. *Human-Computer Interaction.* **12,** 391-438.

[20] Ko, A. J. & Myers, B.A. (2005). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing,* **16**, 41-84.

[21] Lawrance, T., Bellamy, R., Burnett, M. & Rector, K. (2008), Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proceedings of the ACM conference on human factors in computing: CHI 2008.* New York, NY: ACM.

[22] McCabe, T. J. (1976) A complexity measure, *IEEE Transactions on Software Engineering,* **SE-2** (4), 308-320.

[23] Miyake, A. and Shah, P. (eds) (1999), *Models of working memory: Mechanisms of active maintenance and executive control.* New York, NY: Cambridge University Press.

[24] Nystrom, N.A., Urbanic, J., and Savinell, C. Understanding Productivity Through Non-intrusive Instrumentation and Statistical Learning. P-PHEC 2005, San Francisco.

[25] Pirolli, P (2007). *Information foraging theory: Adaptive interaction with information.* Cambridge: Oxford University Press.

[26] Simon, H. A. (1962). The architecture of complexity. *Proceedings of the American Philosophical Society, 106,* 476-482. Philadelphia: American Philosophical Society.

[27] Snir, M. and Bader, D. A. (2003). A framework for measuring Supercomputer productivity. International Journal of High Performance Computing Applications, 18, (4), pp. 417-432.

[28] Thomas, J. C. & Richards, J. T. (2008). Achieving psychological simplicity: Measures and methods to reduce cognitive complexity. In A. Sears & J. Jacko (Eds). *The human-computer interaction handbook: Fundamentals, Evolving Technologies and Emerging Applications.* New York: Erlbaum, 498-507.