Ultra-automation and ultra-autonomy for software engineering management of ultra-large-scale systems

Philip M. Johnson
Collaborative Software Development Laboratory
Department of Information and Computer Sciences
University of Hawai'i
Honolulu, HI 96822
johnson@hawaii.edu

Abstract

"Ultra-Large-Scale Systems: The Software Challenge of the Future" [10] identifies "Engineering Management at Large Scales" as an important focus of research. Engineering management for software typically involves measurement and monitoring of products and processes in order to maintain acceptable levels of important project characteristics including cost, quality, usability, performance, reliability, and so forth. Our research on software engineering measurement over the past ten years has exhibited a trend towards increasing automation and autonomy in the collection and analysis of process and product measures. In this position paper, we extrapolate from our work so far to consider what new forms of automation and autonomy might be required for software engineering management of ULS systems.

1. From PSP to Leap to Hackystat

About ten years ago, I was smitten by the Personal Software Process[1], which appeared to provide an elegant approach to giving software developers empirical, objective, and actionable feedback regarding their activities. I practiced and taught the PSP in its original form for several semesters, and came to the conclusion that all it really lacked was an adequate toolset. Filling out Word documents by hand was simply too much overhead and had the potential to introduce significant mistakes in collection and analysis, regardless of the potential benefits of the information[5].

Thus began Project LEAP, an attempt to reduce the overhead involved in collecting and analyzing PSP-style metrics through tools, as well as enable experimental comparison of the PSP method with variants, such as alternatives to the PROBE estimation method. One outcome of that work was the realization that in many cases, "you can't even get them to push a button" [3]. Put another way, developers don't like to interrupt their flow state to interact with a measurement tool, even if that interaction is reduced to the most simple interaction possible.

Project LEAP was thus superceded by Project Hackystat[2], a new approach in which developers wouldn't have to do *anything* to collect software engineering process and product data. Instead, software "sensors" would be attached to their development tools. These sensors would unobtrusively collect information and send it to a centralized server. The centralized server would aggregate together all of the various kinds of information sent by all of the project team members and analyze it. If anything interesting was discovered, it would send an email off to developers with information about the development metrics and trends to which they might want to pay attention.

Hackystat is now five years old, and consists of over 350,000 lines of code. The set of sensors is extensible and currently includes support for over 30 tools including: IDEs (Eclipse, Emacs, JBuilder, Vim, Visual Studio, Idea), testing (JUnit, CppUnit, Emma, NUnit), build (Ant, Make), configuration management (CVS, Subversion), static analysis (Checkstyle, FindBugs, PMD), bug tracking (Jira, Bugzilla), size metrics for over twenty five programming languages (SCLC, LOCC, CCCC), and management (Microsoft Office, OpenOffice.org).

The low-level data sent by sensors is represented in terms of an extensible set of abstractions called "sensor data types", such as Activity, CodeIssue, Coverage, or FileMetric, which facilitate data consistency and simplify higher level processing.

On the server side, an extensible set of analysis modules process the raw sensor data to create higher-level ab-

stractions that support software development research and management. For example, the Software Project Telemetry module provides support for trend analysis of multiple sensor data streams to aid in-process decision-making [6], the Zorro module provides support for automated recognition of Test Driven Development [8], the MDS module provides support for build process analysis for NASA's Mission Data System project [4], the HPC module supports analysis of high performance computing software development [7], and the CGQM module provides a "continuous" approach to the Goal-Question-Metric paradigm [9].

An organization can use Hackystat to instantiate a tailored system by selecting components from our public repository, and can also augment the public components with proprietary Hackystat components they develop in-house. We host a public server with over 600 registered users, and Hackystat has been installed and used by a variety of research and commercial organizations world-wide.

2. (Way) Beyond Hackystat

Engineering management of ultra-large-scale systems would appear to require not one but two research "order of magnitude" advances over the current approach to software engineering process and product measurement and analysis as embodied by Hackystat. To make this more concrete, let me offer some observations on what Hackystat currently fails to do well from our experience using Hackystat to support the development of the Hackystat software system itself.

One shortcoming of Hackystat is that it does not realize the potential to provide software agents as first class developers. For example, our daily automated build process behaves in many ways like a regular developer: it checks out the code from Subversion, compiles the system, builds and installs test servers, runs the unit tests and other quality assurance tools, and is instrumented with sensors so that the results of these activities are reported to the Hackystat server. Some of our alerts have a crude kind of "intelligence"—for example, we have a build analysis mechanism that analyzes the failure reports, compares them to the Subversion commit data, and generates an email to developers containing its conjectures on the "culprits" who failed the build. These facilities fail to be real "agents" in part because they are not "reactive" in any real sense–both are essentially cron jobs as opposed to entities that actively monitor their environment and react to it.

A second shortcoming of Hackystat is that it always requires humans to "close the loop" in terms of reacting to the data and analyses collected by the sensors. In other words, data can be automatically collected, automatically analyzed, and automatically provided back to developers. But, any process or product *changes* indicated by that data must always be made by developers. If we could become sufficiently confident of the analyses made by Hackystat, then we could "close the loop" by enabling the system (or agents interacting with the system) to make process or product changes in addition to the human developers. For example, certain kind of build failures can be prevented by the developers taking additional actions in their local environment before committing changes. We do not enforce these actions, because they are time-consuming and costly and experienced developers avoid making these mistakes. However, under certain circumstances it would be useful to have the environment recognize the introduction of "novice" developers and impose additional process constraints on them based upon the kinds of build failures that they generate.

A third shortcoming of Hackystat is its inability to understand the nature of the users interacting with it, and make decisions about the kind of information that they need. Hackystat has grown to the point where the range of potential analyses available is bewildering to beginning users, and uncomfortable for veteran users, both of whom often feel that they are probably missing something important in the data. Even though Hackystat accumulates a great deal of behavioral information about its users, it makes no attempt to use this information to create a user model that would enable it to provide tailored information of more relevance to the individual.

3. Ultra Large Scale Software Engineering Process and Product Measurement

My particular interest in ULS is thus on ways to support software engineering process and product measurement and analysis. My intuition is that addressing all of the shortcomings noted above would only get Hackystat to the "large scale". What would it mean to jump up yet another level in scale and complexity?

One possibility is that the system would have to have new levels of autonomy in terms of the ways it makes connections between data. For example, current Hackystat servers only perform analyses on Projects that are explicitly defined by users. In ULS, one would assume that the system would not only need to be able to decide for itself what constitutes a "Project", but also autonomously establish connections with other Hackystat servers in order to share data. This, in turn, implies the need for a concept of "locality" with respect to software development so that servers could request data "near" to them from a measurement point of view.

A second possibility is that the system would need to be able to define its own sensors, sensor data types, and analyses in response to emergent conditions, or at least be able to autonomously instantiate generic sensors, sensor data types, and analyses with parameters based upon the context at

hand.

While these possibilities form exciting research directions for ULS engineering management, they also raise the question of evaluation: how can one perform an initial evaluation of a ULS mechanism outside the context of actual ULS systems in order to assess their feasibility and effectiveness? In other words, what would constitute a useful "laboratory model" of a ULS system: one that preserves some of the important characteristics of ULS systems while eliminating other scales in order to support experimentation?

With respect to engineering management, I believe that current open source system projects (such as the Linux operating system, the Eclipse IDE, the Apache web server, and even the Hackystat Framework) can be potentially useful "models" for evaluation of certain ULS engineeering management innovations. Systems such as Linux, Eclipse, Apache, and Hackystat share a diverse user and developer community that are simultaneously extending these frameworks in orthogonal, potentially incompatible directions. As open source, they can support experimentation with new, decentralized paradigms for measurement and management. Finally, they provide a level of transparency in product and process that can facilitate objective, scientific evaluation of the strengths and weaknesses of new approaches.

The ULS program provides a way to push the research trajectory of software engineering measurement from its humble beginnings as Word templates to self-adapting, autonomous federations of systems. I look forward to an opportunity to gather together with other researchers to learn from their perspectives and refine my own.

References

[1] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, New York, 1995.

[2] P. M. Johnson. Hackystat Framework Home Page. http://www.hackystat.org/.

[3] P. M. Johnson. You can't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering. In *The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications*, Nashville, TN, December 2001.

[4] P. M. Johnson. The Hackystat-JPL configuration: Overview and initial results. Technical Report CSDL-03-07, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, October 2003.

[5] P. M. Johnson and A. M. Disney. A critical analysis of PSP data quality: Results from a case study. *Journal of Empirical Software Engineering*, December 1999.

[6] P. M. Johnson, H. Kou, M. G. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita. Improving software development management through software project telemetry. *IEEE Software*, August 2005.

[7] P. M. Johnson and M. G. Paulding. Understanding HPCS development through automated process and product measurement with Hackystat. In *Second Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, February 2005.

[8] H. Kou and P. M. Johnson. Automated recognition of low-level process: A pilot validation study of Zorro for test-driven development. In *Proceedings of the 2006 International Workshop on Software Process*, Shanghai, China, May 2006.

[9] C. Lofi. Continuous GQM: An automated framework for the goal-question-metric paradigm. M.S. Thesis CSDL-05-09, Department of Software Engineering, Fachbereich Informatik, Universitat Kaiserslautern, Germany, August 2005.

[10] L. Northrup. Ultra-large-scale systems: The software challenge of the future. Technical report, Software Engineering Institute, 2006.