Improving Software Development Management through Software Project Telemetry

Philip M. Johnson, Hongbing Kou, Michael Paulding, Qin Zhang, Aaron Kagawa, and Takuya Yamashita, *University of Hawaii*

Software project telemetry facilitates local, in-process decision making. Hackystat, an open source reference framework for this new approach, offers easy implementation.

Conventional wisdom in the software engineering research community says that metrics can make project management more effective.[1] Software metrics range from internal product attributes such as size, complexity, and modularity to external process attributes such as effort, productivity, and reliability. Software metrics proponents quote theorists and practitioners from Galileo's "What is not measurable, make measurable"[2] to Tom DeMarco's "You can neither predict nor control what you cannot measure."[3]

Despite metrics' theoretical potential, effectively applying them appears to be far from mainstream in practice. For example, a recent case study of more than 600 software professionals revealed that only 27 percent viewed metrics as "very" or "extremely" important to their software project decision-making process.[4] The study also revealed that most respondents attempted to use metrics only for cost and schedule estimation.

Practitioners face various barriers in applying metrics (see the sidebar "Explaining the Gap between Software Metrics Theory and Practice"). It's no wonder that many practitioners find it daunting to apply best practices to their own situation. Indeed, the agile community generally argues against model-based metrics applications, promoting softer metrics for decision making.[5] Fortunately, creating predictive models based on historical project data isn't the only possible way to apply software metrics to project management. Our team at the Collaborative Software Development Laboratory has developed a new telemetry-based approach.

Software project telemetry

According to the *Encyclopedia Britannica*, telemetry is a "highly automated communications process by which measurements are made and other data collected at remote or inaccessible points and transmitted to receiving equipment for monitoring, display, and recording." Perhaps the highest-profile telemetry user is NASA, which has been using it since 1965 to monitor operations from the early Gemini missions to the modern Mars Rover flights. At NASA's Mission Control

0740-7459/05/$20.00 © 2005 IEEE

Explaining the Gap between Software Metrics Theory and Practice

The *Software Metrics Best Practices* report[1] reveals a substantial gap between software metrics theory and its actual implementation in practice. Why might this be? One hypothesis is that most practitioners are simply uninformed. Perhaps if they would subscribe to journals and attend conferences on software metrics, they could immediately implement current best practices and just as quickly improve their project management decision making.

All of us, theorists and practitioners alike, can always benefit from additional education. However, an alternative explanation is that perhaps the metrics methods that theorists use yield results that developers can't easily translate into practice. To see why, consider that much metrics research involves the following basic method:

1. Collect a set of process and product measures (such as size, effort, known defects, and complexity) for a set of completed software projects.
2. Generate a model that fits this data.
3. Claim that this model can now be used to predict future project characteristics.

For example, a model might predict that a future project of size *S* will require *E* person-months of effort; another model might predict that the implementation of a module with complexity *C* will be prone to defects with density *D*. Unfortunately, practitioners face several barriers to adopting these predictive, model-based metrics approaches.

First, to use the model unchanged, practitioners must confirm that the set of projects the researcher uses to generate the model is similar to their current projects. This is the context problem: unless the context associated with the process and project data in the model is sufficiently similar to the context associated with a practitioner's projects, the model's outputs might not apply to the practitioner.

Next, practitioners must also confirm that their future projects' context will remain similar to their previous ones. If practitioners can't meet those two conditions, they must recalibrate the model at some point. This involves replicating the model-building method within a practitioner's organization—with the risk that the resulting model won't work or that the organization might change yet again, rendering future project contexts different from those used to calibrate the model.

Finally, practitioners must assess the cost of metrics collection and analysis required by the model and ensure that the metrics and their interpretation are useful enough to justify these costs.

Reference

1. P. Kulik and M. Haas, *Software Metrics Best Practices 2003*, tech. report, Accelera Research, 2003.

Center, for example, dozens of specialists monitor telemetry data from sensors attached to a space vehicle and its occupants. They use this data for many purposes, including early warning of anomalies indicating problems, better insight into the mission's status, and gauging the impact of incremental course or mission adjustments.

We define *software project telemetry* as a style of software metrics definition, collection, and analysis with the following essential properties:

- *The data is collected automatically by tools that regularly measure various characteristics of the project development environment*. In other words, software developers work in a location that's remote or inaccessible to manual metrics collection activities. This contrasts with software metrics data that requires human intervention or developer effort to collect, such as Personal Software Process or Team Software Process metrics.[6]
- *The data consists of a stream of time-stamped events where the time-stamp is significant for analysis*. Software project telemetry data focuses on the changes over time in measurements of processes and products during development. This contrasts, for example, with Cocomo,[7] where the precise time at which calibration data is collected generally isn't relevant.
- *Both developers and managers can continuously and immediately access the data*. Telemetry data isn't hidden away in some obscure database that the software quality improvement group guards. All project members can examine and interpret it.
- *Telemetry analyses exhibit graceful degradation*. Although complete telemetry data provides the best project management support, the analyses shouldn't be brittle: they should still provide value even if complete data over the entire project's lifespan isn't available. For example, telemetry collection and analysis should provide decision-making value even if

Software project telemetry provides a measurement infrastructure that isn't focused on a particular long-range target.

these activities start midway through a project.

- *Analysis includes in-process monitoring, control, and short-term prediction.* Telemetry analyses represent the current project state and how it changes at various timescales; so far, days, weeks, and months are useful scales. The simultaneous display of multiple project state values and how they change over the same time periods allows opportunistic analyses—the emergent knowledge that one state variable appears to co-vary with another in the current project context.

Software project telemetry enables an incremental, distributed, visible, and experiential approach to project decision making. For example, if you found that both complexity and defect-density telemetry values were increasing, you could take corrective action (for example, by simplifying overly complex modules) to try to decrease the defect-density telemetry values. You could also monitor other telemetry data to see if such simplification has unintended side effects (such as performance degradation). Project management using telemetry thus involves cycles of hypothesis generation ("Does module complexity correlate with defect density?"), hypothesis testing ("If I reduce module complexity, will defect density decrease?"), and impact analysis ("Do the process changes required to reduce module complexity produce unintended side effects?"). Finally, software project telemetry supports decentralized project management: because all project members can access telemetry data, it helps both developers and managers engage in these management activities.

Software project telemetry is related to in-process software metrics, such as work done on software testing management.[8] However, such work tends to focus on a narrow range of measures and management actions related to testing and, as a result, is amenable to manual data collection and analysis. Telemetry's broader scope necessitates automated collection and analysis, with correspondingly broader management decision-making support.

Software project telemetry also relates in interesting ways to both the Capability Maturity Model Integration and Agile methods. The CMMI, a revision of the original CMM, incorporates lessons learned and supports more modern, iterative development processes. In an article on transitioning from CMM to CMMI, Walker Royce asserts the importance of "instrument[ing] the process for objective quality control. Lifecycle assessments of both the process and all intermediate products must be tightly integrated into the process, using well-defined measures derived directly from the evolving engineering artifacts and integrated into all activities and teams."[9] Software project telemetry appears well suited to the CMMI vision for process improvement.

Agile method proponents are traditionally suspicious of conventional process and product measurements and technologies for project decision making. Jim Highsmith frames the problem as follows: "Agile approaches excel in volatile environments in which conformance to plans made months in advance is a poor measure of success. If agility is important, one characteristic we should measure is that agility. Traditional measures of success emphasize conformance to predictions (plans). Agility emphasizes responsiveness to change. So there is a conflict because managers and executives say that they want flexibility, but then they still measure success based on conformance to plans."[10] Software project telemetry provides a measurement infrastructure that isn't focused on achieving a particular long-range target but on process and product measurements that support adaptation and improvement.

Support for telemetry: Project Hackystat

For several years, we've been designing, implementing, and evaluating tools and techniques to support a telemetry-based software project management approach as part of Project Hackystat. Figure 1 illustrates the system's overall architecture. Developers instrument the project development environment by installing Hackystat sensors into various tools, such as their editor, build system, and configuration management system. Once installed, the Hackystat sensors unobtrusively monitor development activities and send process and product data to a centralized Web service. Project members can log in to the Web server to see the collected raw data and run analyses that integrate and abstract the raw sensor data streams into telemetry. Hackystat also lets project members configure alerts that watch for specific conditions in the telemetry stream and sends an

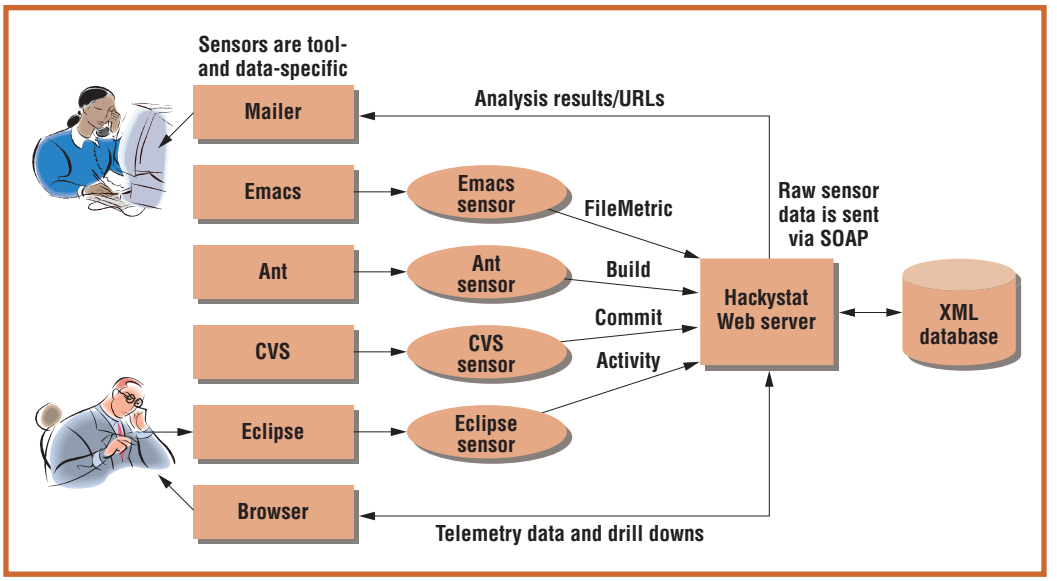


Figure 1. Hackystat's basic architecture. Sensors are attached to tools that developers directly invoke (such as Eclipse or Emacs) and tools that developers implicitly manipulate (such as Concurrent Versions System or an automated build process using Ant).

email when these conditions occur.

Hackystat supports the following general classes of software project telemetry:

- *Development telemetry* is data that Hackystat gathers by observing the project developers' and managers' behavior based on their tool usage. It includes information about the files they edit, the time they spend using various tools, the changes they make to project artifacts, and the sequences of tool or command invocations. Hackystat provides sensors for collecting development telemetry from editors such as Eclipse or Emacs, Office applications such as Word or Frontpage, configuration management tools such as CVS, and issue management tools such as Jira.
- *Build telemetry* is data gathered by observing the results of tools invoked to compile, link, and test the system. Hackystat can gather such data from build tools such as Ant, Make, or CruiseControl, testing tools such as JUnit or CppUnit, and size and complexity tools such as LOCC.
- *Execution telemetry* is data that Hackystat gathers by observing the system's behavior as it executes. Hackystat provides sensors for collecting execution telemetry from tools that test for load or stress, such as JMeter.
- *Usage telemetry* is data that Hackystat gathers by observing the users' interaction with the system. This data includes user behavior such as frequency, types, and sequences of command invocations during a given time period in a given subsystem.

For a description of specific sensors and data types that Hackystat supports, see the "Sensors and Sensor Data Types" sidebar.

The path from sensors to the telemetry report involves several steps. Hackystat sensors collect raw data by observing behavior in various client tools and then send it to the Hackystat server, which persists the data in an XML-based repository. Hackystat analysis mechanisms then abstract this raw data into DailyProjectData instances, which can involve synthesizing sensor data from multiple group members or sensors into a higher-level representation of process or product characteristics for a given project and day. Sets of DailyProjectData instances are then manipulated by Reduction Functions, which emit a sequence of numerical telemetry values for a given project at a timescale of days, weeks, or months.

The previous steps occur automatically as part of the software project telemetry implementation. An important benefit of Hackystat is its explicit support for the exploratory nature of telemetry-based decision making. We've designed a Telemetry Display Language that we use with the Hackystat Web server to interactively define telemetry streams and specify how practitioners should compose them together into charts and reports for pres-

Sensors and Sensor Data Types

Hackystat provides an extensible architecture with respect to both sensors, the software plug-ins associated with development tools, and sensor data types, which describe a given type of raw metric data's structure. This mapping isn't one-to-one: for example, the Eclipse sensor can send Activity, File-Metric, and Review sensor data types, and both Integrated Development Environment and Size metric sensors can collect the File-Metric sensor data type. The following lists describe the range of available sensors and sensor data types; each organization can decide whether to enable these facilities or implement their own custom extensions to facilitate their needs.

Hackystat sensors are currently implemented for the following tools:

- interactive development environments, including Eclipse, Emacs, JBuilder, Vim, and Visual Studio;
- office productivity applications, including Excel, Word, PowerPoint, and Frontpage;
- build tools, including Ant and the Unix command line;
- size measurement tools, including CCCC and LOCC;
- testing tools, including JUnit and JBlanket;
- configuration management, including Concurrent Versions System and Harvest; and
- defect tracking tools, including Jira.

Hackystat sensor data types include

- Activity, which represents data concerning the active time developers spend in their IDE;
- BufferTransition, which represents the sequence of files that developers visit;
- Review, which represents data on review issues generated during code or design inspections;
- FileMetric, which represents file size information;
- Build, which represents data about software build occurrences and outcomes;
- Perf, which represents data about the occurrence and outcome of performance analysis activities such as load testing;
- CLI, which represents data about command line invocation occurrences;
- UnitTest, which represents data about unit test invocation occurrences and outcomes;
- Coverage, which represents data about the coverage that unit testing activities obtain;
- Commit, which represents data about developers' configuration management commit events; and
- Defect, which represents information about posting defect reports to defect tracking tools by developers or users.

Hackystat is an open source system that's freely available for download and use. We encourage interested theorists and practitioners to visit the developer services Web site at www.hackystat.org for access to source code, binaries, and documentation. To try out Hackystat, we also maintain a public server at http://hackystat.ics.hawaii.edu.

entation to developers and managers.

Figure 2 shows an example telemetry report. This report illustrates the relationship between aggregate code churn (the lines all project members add and delete from the CVS repository) and aggregate build results (the number of build attempts and failures on a given day via the Ant build tool). Telemetry reports are always defined without reference to a specific project or time interval. We wait to specify the project and its time interval until we generate the report. Thus, project members can run this telemetry report over differing sets of days or change the timescale to weeks from months to see if different trends emerge from these alternative perspectives. In addition, once a telemetry report is defined, members of other projects on this server could use it to see if it adds decision-making value to their project management activities.

Telemetry in practice: Managing integration-build failure

As a concrete example of software project telemetry in action, we're currently using it to investigate and improve our own daily (integration) build process. Hackystat consists of approximately 95 KLOC, organized into approximately 30 modules, with five to 10 active developers. The CVS configuration management system stores the sources, so developers can check out the latest version of the sources associated with any given module and commit their changes when finished. Developers rarely compile, build, and test against the entire code base; instead, they select a subset of the modules relevant to their work. An automated nightly build process compiles, builds, and tests the latest committed code for all modules and sends email if the build fails. We can also invoke this integration build manually.

At the end of 2004, we discovered that our integration-build failure rate was significant. For the 300 daily integration-build attempts during that year, the build failed on 88 days with a total of 95 distinct build errors. This high failure rate substantially impacted our productivity. Each failure generally required one or more team members to stop concurrent development, diagnose the problem, determine who was responsible for fixing it, and often wait until the corrections were committed before checking out or committing additional code.

To reduce the integration-build failure rate in 2005, we needed to better understand how, when, and why the previous builds had failed in 2004. To do this, we embarked on a series of analyses involving several Hackystat sensor data streams: Active Time, Commit, Build, and Churn. This revealed many useful insights about our build process:

- We could partition the 95 distinct build errors into six categories: coding style error (14), compilation error (25), unit test error (40), build script error (8), platform-related error (3), and unknown error (5).
- We found substantial differences between experienced and new developers with respect to integration-build failures. For example, the least experienced developer had the highest integration-build failure rate—an average of one build failure per four hours of Active Time. In contrast, more experienced developers averaged one build failure per 20 to 40 hours of Active Time.
- The two modules with the most dependencies on other modules also had the two highest numbers of build failures, and together they accounted for almost 30 percent of the failures.
- We found that the 88 days with build failures had, on average, a statistically significant greater number of distinct module commits than days without build failures.
- We found (somewhat unexpectedly) that there was no relationship between build failure and the number of lines of code committed or the amount of Active Time before the commit. In other words, whether you worked five minutes or five hours before committing, or whether you changed five or 500 lines of code, didn't change the odds of causing a build failure.

These findings yield numerous hypotheses regarding ways of reducing integration-build failure, including increased support for new developers (such as pair programming) and refactoring modules to reduce coupling and frequent multimodule commits. The most provocative hypothesis, however, is that 82 percent of the integration failures in 2004 could have been prevented if the developers had run a full system compile and test before committing their changes.

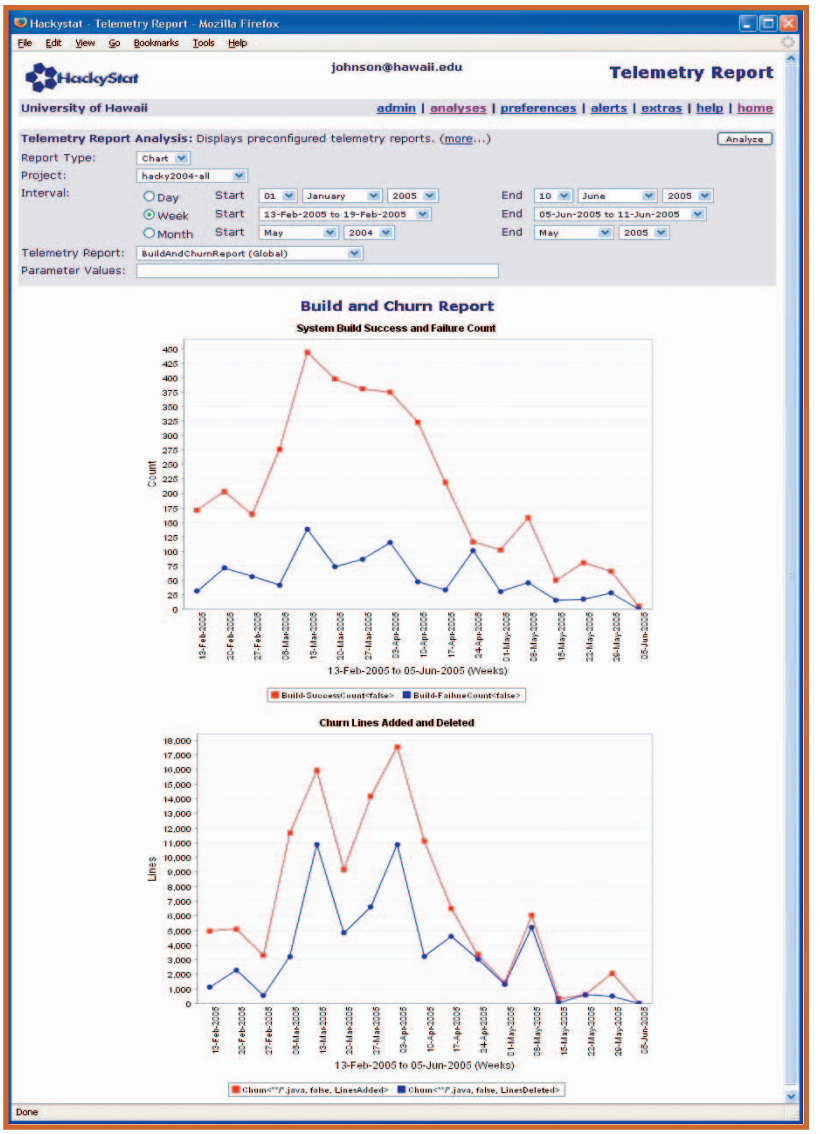


Figure 2. A telemetry report that compares code churn (lines added and deleted) to build results (number of build attempts and failures).

The most simple—and most intrusive—process improvement following from this hypothesis is to require all developers to run a full system compile and test locally before every commit. However, even though some untested commits result in an integration-build failure, many other untested commits do not. Given that a full compile and test can take over 30 minutes and that multiple developers often perform multiple commits per day, this process improvement's productivity cost could actually exceed the benefits of reducing the level of integration-build failures. Other generic changes, such as moving to continuous integration, also tend to move build failure costs around without necessarily reducing them.

Understanding why any given integration build succeeds or fails requires multiple forms of process and product information.

Our current approach to managing integration-build failure comes from recognizing that the decision to invest the time to perform a full build and test before any particular commit depends on many factors. The decision involves developer familiarity with the system, the actual changes made to the code, and other developers' commits. To improve our productivity, we need to give developers tools and feedback to better decide when to precede a specific commit with a full build and test. To assess whether the feedback is working, we can use software project telemetry.

Our analyses demonstrate that understanding why any given integration build succeeds or fails requires multiple forms of process and product information, including the occurrence of local builds, the integration build's success or failure, the failure type, the modules that were committed, the developers responsible, the dependency relationships among modules, and the Active Time associated with the work prior to commits. Unfortunately, we haven't discovered any analytical models that can automate the decision-making process and tell developers before they commit whether they'll need full or partial testing. However, using software project telemetry, we can track the absolute level of integration-build failures over time, as well as the number of failures due to any particular cause, and even the number of failures that we could have prevented with a full build and test.

Using Hackystat's alert mechanism, we can give developers a detailed summary of the process and product state, including the factors we've identified as relevant, whenever an integration-build failure occurs. Our hypothesis is that, given appropriate feedback, each developer will naturally learn over time to be more sophisticated in deciding when to perform a full build and test. New developers might quickly learn to do it almost always, while more experienced developers might begin to recognize more complex indicators. One of our project goals for 2005 is to test this hypothesis and measure the results using integration-build failure telemetry streams.

The Telemetry Control Center

For even a moderately large, complex project, the number of possible telemetry charts and reports quickly explodes. For example, almost a dozen different sensor data streams monitor the Hackystat development project across 30 modules, from five to 10 active developers. Given that each telemetry stream can be composed from one or more sensor data streams, project modules, and developers, you can see the problem: which of the literally thousands of possible charts should we be monitoring?

Our development group decided to address this problem by creating a new interface to the telemetry data that would let us passively monitor telemetry in a way that a standard Web browser wouldn't allow. We call this interface the Telemetry Control Center (see figure 3).

The TCC consists of a standard PC with a multihead video display card that's attached to nine 17-inch liquid crystal display panels, mounted on the wall in our laboratory. We implemented a new client-side software system called the TelemetryViewer, which periodically requests telemetry reports from the Hackystat server, retrieves the resulting image file, and displays it on screens. The TelemetryViewer reads in an XML configuration file at startup, which tells it which reports to retrieve, where to display them, and how long to wait before retrieving the next set of reports. The TelemetryViewer's default behavior is to automatically and repeatedly cycle through the set of telemetry scenes in the XML configuration file.

The TCC frees us from the "tyranny of the browser" by making a sequence of telemetry report sets continuously available without any developer action. It also lets us more easily look for relationships between telemetry streams, because the system can display nine telemetry reports simultaneously. Finally, it provides a new kind of passive awareness about the project's state to all developers; rather than having to decide to generate a report or wait for a weekly project update meeting, developers can simply glance at the TCC whenever they're passing though the lab to get a perspective on the state of development. Individuals can still get all TCC reports on their local workstations if they so desire, although without the simultaneous display.

Lessons learned

Our first lesson learned is that software project telemetry can help support project management decision making. In addition to our work on integration-build failure, telemetry data has also revealed to us a recent, subtle

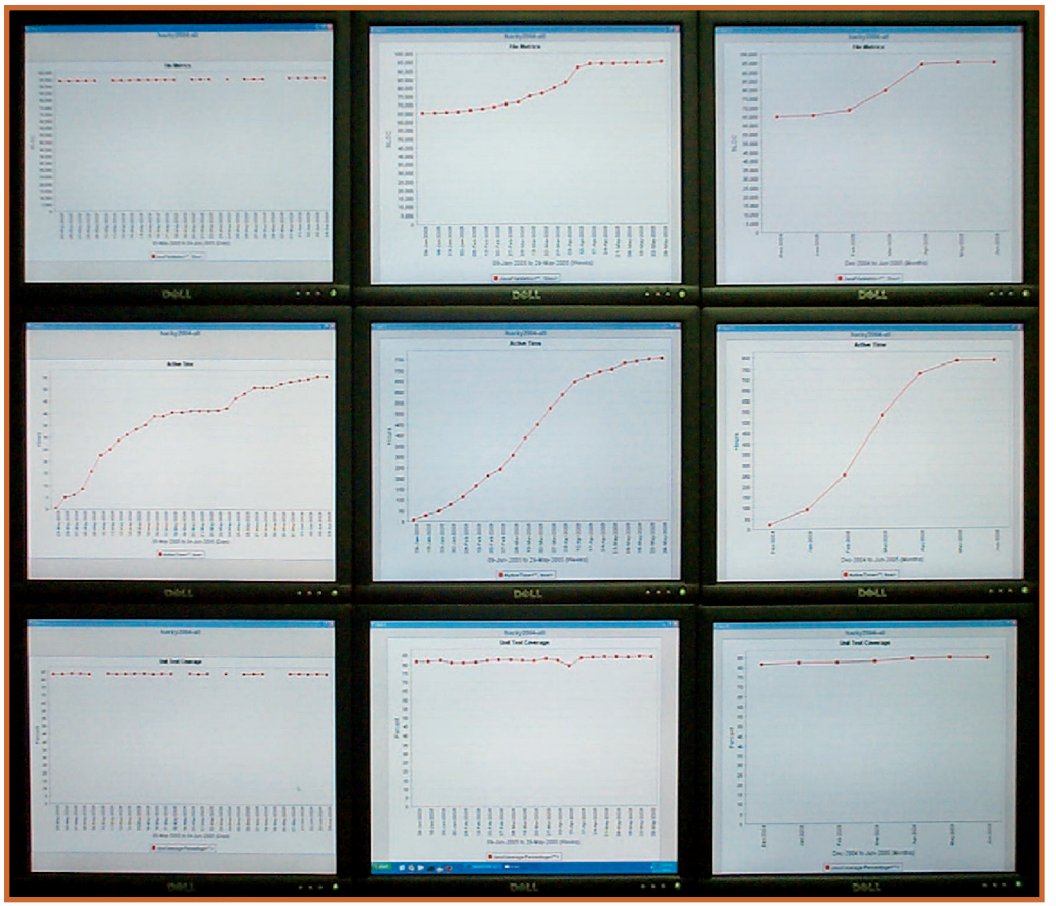


Figure 3. The Telemetry Control Center, which shows one scene consisting of nine telemetry reports. The associated TelemetryViewer software controls the TCC by automatically cycling through a set of scenes at a predefined interval. This telemetry viewer is configured to show a dozen separate scenes, each displayed for two minutes.

slide in testing coverage over the past six months that has co-occurred with two significant refactoring episodes (and resulting code churn). As a result, we're allocating additional effort to software review with a focus on assessing new modules' test quality. We hope that this project management decision will eventually reverse the declining coverage trend.

Hackystat provides an open source reference framework for software project telemetry, but it isn't the only technology available. Commercial measurement tools can also provide infrastructure support, or your organization could decide to develop technology in-house. The key issue is to preserve software project telemetry's essential properties.

We've learned that having an automated daily build mechanism adds significant value to software project telemetry. It provides not only a convenient hook into which you can add sensors to reliably obtain daily product measures information but also a kind of heart beat for the development project that makes all the metrics more comparable, accessible, and current.

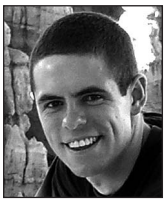
As with any measurement approach, you must consider social issues. It's possible to misinterpret and misuse software project telemetry data. For example, telemetry data is intrinsically incomplete with respect to measuring effort. Hackystat implements a measure called Active Time, which is the time developers and managers spend editing files related to a given project in tools such as Eclipse, Word, or Excel. However, many legitimate and productive activities, including meetings, email, and hallway conversations, are outside the telemetry-based measurement's scope. Telemetry can't measure effort in its broadest sense, so a project member with little Active Time could still be contributing significantly to the project. Indeed, some organizations might decide not to collect measures such as Active Time, simply because it's susceptible to misinterpretation and abuse. For an excellent analysis of these and other forms of "measurement dysfunction," see Robert Austin's *Measuring and Managing Performance in Organizations*.[11]

Adopting a software project telemetry approach to measurement and decision making

About the Authors

Philip Johnson is a professor of information and computer sciences at the University of Hawaii and the director of its Collaborative Software Development Laboratory. His research interests include software engineering and computer-supported collaborative work. He received his PhD in computer science from the University of Massachusetts. He's a member of the IEEE and the ACM. Contact him at the Collaborative Software Development Laboratory, 1680 East-West Rd., POST 307, Univ. of Hawaii, Honolulu, HI 96822; johnson@hawaii.edu.

Hongbing Kou is a PhD student at the University of Hawaii and a researcher in its Collaborative Software Development Laboratory. His research interests include software engineering and software process modeling, including agile methods. He received his MS in computer science from the University of Hawaii. Contact him at the Collaborative Software Development Laboratory, 1680 East-West Rd., POST 307, Univ. of Hawaii, Honolulu, HI 96822; hongbing@hawaii.edu.



Michael Paulding is a PhD student at the University of Hawaii and a researcher in its Collaborative Software Development Laboratory. His research interests include software engineering practices and productivity studies of high-performance computing systems. He received his BS in computer science and engineering from Bucknell University. Contact him at the Collaborative Software Development Laboratory, 1680 East-West Rd., POST 307, Univ. of Hawaii, Honolulu, HI 96822; mpauldin@hawaii.edu.

Qin (Cedric) Zhang is a PhD student at the University of Hawaii and a researcher in its Collaborative Software Development Laboratory. He received his MA in economics and his MS in information and computer sciences from the University of Hawaii. Contact him at the Collaborative Software Development Laboratory, 1680 East-West Rd., POST 307, Univ. of Hawaii, Honolulu, HI 96822; qzhang@hawaii.edu.

Aaron Kagawa is a master's student in information and computer sciences at the University of Hawaii, a graduate research assistant, and a member of the Collaborative Software Development Laboratory. His research interests include software project management, software quality, and software inspections. He received his BS with Honors in information and computer sciences from the University of Hawaii. Contact him at the Collaborative Software Development Laboratory, 1680 East-West Rd., POST 307, Univ. of Hawaii, Honolulu, HI 96822; kagawaa@hawaii.edu.

Takuya Yamashita is a master's student at the University of Hawaii and a researcher in its Collaborative Software Development Laboratory. His research interests include software engineering and automated support for code inspection. He received his BA in law from Hosei University and his BA in economics from the University of Hawaii at Hilo. Contact him at the Collaborative Software Development Laboratory, 1680 East-West Rd., POST 307, Univ. of Hawaii, Honolulu, HI 96822; takuyay@hawaii.edu.

tends to exert a kind of gravitational force toward increased use of tools for managing process and products. For example, a small development team might begin by informally managing tasks and defects using email or index cards. As the team adopts telemetry-based decision making, it will inevitably want to relate development process and product characteristics to open tasks and defect severity levels. However, it won't be able to unless it moves to an issue management tool such as Bugzilla or Jira that can support sensor-based measurement.

Does software project telemetry provide a silver bullet that solves all problems associated with metrics-based software project management and decision making? Of course not. While software project telemetry does address certain problems inherent in traditional measurement and provides a new approach to more local, in-process decision making, it provides its own issues that future research and practice must address.

Telemetry data's decision-making value is only as good as the data that sensors can obtain. Clearly, some threshold exists for sensor data beneath which the telemetry's decision-making value is compromised. But what is this threshold, and how does it vary with the kinds of decision making the development group requires? What sets of sensors and sensor data types are best suited to what project development contexts?

What are telemetry-based data's intrinsic limitations? A good way to investigate this question involves qualitative, ethnographic research, in which a researcher trained in these methods observes a software development group to learn what kinds of information relevant to project management decision making occur outside the realm of telemetry data.

While manual investigation of telemetry streams and their relationship to each other is certainly an important and necessary first step, the sheer number of possible relationships and interactions means that only a small percentage of them can be inspected and monitored manually on an ongoing basis. An intriguing future direction is exploring whether using data mining and clustering algorithms can reveal relationships in the telemetry data that manual exploration might not discover.

What are the costs associated with initially setting up software project telemetry using a system like Hackystat? Unfortunately, we can't use Hackystat to measure the effort involved with installing Hackystat. Furthermore, some organizations will require development of new sensors or analyses not available in the standard distribution. Better understanding of these costs will aid adoption of this technology and method.

Acknowledgments

Partial support for Project Hackystat and Software Project Telemetry was provided by NSF grant CCR-0234568, NASA, and Sun Microsystems.

References

1. N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, Thomson Computer Press, 1997.
2. L. Finkelstein, "What Is Not Measurable, Make Measurable," *Measurement and Control*, vol. 15, 1982, pp. 25–32.
3. T. DeMarco, *Controlling Software Projects*, Yourdon Press, 1982.
4. P. Kulik and M. Haas, *Software Metrics Best Practices 2003*, tech. report, Accelera Research, 2003.
5. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.
6. W.S. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, 1995.
7. B. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.
8. S.H. Kan, J. Parrish, and D. Manlove, "In-Process Metrics for Software Testing," *IBM Systems J.*, vol. 40, no. 1, 2001, pp. 220–241.
9. W. Royce, "CMM vs. CMMI: From Conventional to Modern Software Management," *Rational Edge*, Feb. 2002; www-106.ibm.com/developerworks/rational/library/content/RationalEdge/archives/feb02.html.