Requirement and Design Trade-offs in Hackystat: An In-Process Software Engineering Measurement and Analysis System

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

Abstract

For five years, the Hackystat Project has incrementally developed and evaluated a generic framework for in-process software engineering measurement and analysis (ISEMA). At least five other independent ISEMA system development projects have been initiated during this time, indicating growing interest and investment in this approach by the software engineering community. This paper presents 12 important requirement and design trade-offs made in the Hackystat system, some of their implications for organizations wishing to introduce ISEMA, and six directions for future research and development. The three goals of this paper are to: (1) help potential users of ISEMA systems to better evaluate the relative strengths and weaknesses of current and future systems, (2) help potential developers of ISEMA systems to better understand some of the important requirement and design trade-offs that they must make, and (3) help accelerate progress in ISEMA by identifying promising directions for future research and development.

1. Introduction

Most software engineers will agree that measurement can be useful in software development. The disagreements begin when deciding what, when, where, how, and why to measure. What to measure can range from process measures such as build failure rate to product measures such as the size of the system. When to measure can range from in-process measures that require daily or hourly data collection, to out-of-process measures that are collected, for example, after a development project is done as part of a post-mortem. How to measure can range from manual techniques that require a software process group to collect and analyze the data, to automated techniques that require no human involvement at all for collection and analysis (but might still require human involvement for interpretation of the analyses and subsequent decision-making.) Finally, why to measure can range from the building of predictive models to estimate future cost or quality, to assessment of current project characteristics.

Since 2001, we have been developing and evaluating an open source, extensible application framework called Hackystat for in-process software engineering measurement and analysis (ISEMA). The client-server systems resulting from instantiation of the framework enable developers to attach small software plugins called "sensors" to their development tools which unobtrusively collect and send low-level data about their behavior and/or results to a Hackystat web application using SOAP. The set of sensors is extensible and currently includes support for IDEs (Eclipse, Emacs, JBuilder, Vim, Visual Studio), testing (JUnit, CppUnit, Emma), build (Ant, Make), configuration management (CVS, Subversion), static analysis (Checkstyle, FindBugs, PMD), bug tracking (Jira), 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 abstractions 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 [7], the Zorro module provides support for automated recognition of Test Driven Development [10], the MDS module provides support for build process analysis for NASA's Mission

Data System project [5], the HPC module supports analysis of high performance computing software development [8], the CGQM module provides a "continuous" approach to the Goal-Question-Metric paradigm [11], and the Course module supports software engineering education [6]. An organization can use Hackystat to instantiate a tailored ISEMA system by selecting components from our public repository, and can also augment the public components with proprietary Hackystat components they develop in-house.

When we started the Hackystat Project, we had the idealistic (and naive) goal of designing a truly "generic" ISEMA framework, one that would provide appropriate infrastructure to any organization desiring in-process software engineering measurement and analysis. After five years of research and development, we have learned that while Hackystat can be effectively applied to a range of problems, the domain of in-process software engineering measurement and analysis is much too broad for a "one size fits all" solution. Indeed, over the past five years, at least five other ISEMA system development projects have been initiated, including EPM [16], 6th Sense Analytics [4], PROM [15], ECG [13], and SUMS [12]. On the one hand, this surge of activity by the software engineering community appears to validate the utility and potential of ISEMA systems. On the other hand, if Hackystat was truly generic, why were these other projects even started?

Over the course of its development, Hackystat has had over forty public releases, undergone seven major architectural revisions, been used by hundreds of developers, and grown to over 300,000 lines of code. The system and its architecture appears to be relatively mature and stable. Our experiences as developers and the feedback we have received from our users reveal that the requirement and design decisions made during development of an ISEMA system entail fundamental trade-offs along the dimensions of usability, genericity, simplicity, marketability, and performance. We believe that the essential nature of these trade-offs is an important reason for the rise of a community of ISEMA systems. For example, the decision to make Hackystat extensible with respect to sensors, sensor data types, and analyses also makes Hackystat more complicated to install, use, and document than an ISEMA tool like SUMS, which implements a single, operating system-level "sensor" and sensor data type.

This paper presents results from our first five years of research on ISEMA, including a description of 12 important requirement and design trade-offs present in Hackystat, a discussion of how these trade-offs influence the buy-vs-build decision, and implications for an ISEMA research agenda. We present this information in hopes that it will help potential users of ISEMA systems to better evaluate the relative strengths and weaknesses of current and future systems, help potential developers of ISEMA systems to better understand some of the important requirement and design trade-offs that they must make, and finally, help accelerate progress in ISEMA by identifying promising directions for future research and development.

While we will refer to other ISEMA systems during the presentation of the trade-offs, and in fact provide a brief overview of them in the next section, this paper is not a comparative analysis of current ISEMA systems. As developers of the Hackystat system, it would be very hard for us to provide a truly unbiased comparison of the various approaches. Furthermore, while we have almost complete knowledge about the current design and history of the Hackystat system, our knowledge of other systems is limited to externally published documentation. Thus, we believe that we can make the greatest contribution to the community by providing new insights about our own system and our experiences with it, and leaving comparative analysis as an activity for an unbiased third party. That said, the next section provides a brief introduction to ISEMA systems to outline the various approaches underway.

2. ISEMA Systems

ISEMA is a relatively recent approach to software engineering measurement. The more traditional approach might be termed "out of process" measurement, in which data is collected about a set of previously completed projects and used to make predictions about future as-yet-unstarted projects. One of the most successful applications of out of process measurement is COCOMO [3], in which data about the cost, size, and characteristics of previously developed systems is used to produce a predictive model that provides estimates of cost and time for new projects based upon various parameters. Such an approach is out of process since the system is typically used after the completion of old projects but before the initiation of new projects. Other non-ISEMA approaches to software engineering measurement and analysis include PSP/TSP, the IFPUG function point data repository, and the NASA/SEL metrics repository. The ISEMA approach, in contrast, accumulates data about a current project in order to provide feedback and decision-making value back into the very same project. In addition to Hackystat, current ISEMA systems include PROM, EPM, Sixth Sense Analytics, ECG, and SUMS.

PROM. The Professional Metrics (PROM) system [15] is sponsored by the Center for Applied Software Engineering at the Free University of Bolzano-Bozen. PROM supports an approach similar to Hackystat, in which plugins unobtrusively monitor development activities and send process and product data to a centralized server for analysis. PROM provides plugins for Microsoft Office, OpenOffice, Eclipse, Visual Studio, JBuilder, NetBeans, and IntelliJ Idea. It can extract code metrics for C, C++, C#, and Java.

Finally, the Trace tool can support tracking of operating system calls. PROM has been used in case studies of agile methodologies, open source tool evaluation, and knowledge database integration.

EPM. The Empirical Project Monitor (EPM) system [16] is sponsored by the EASE Project, which is an academic-industrial alliance that includes the Nara Institute of Science and Technology, Osaka University, NTT, and Hitachi. EPM does not use a sensor-based approach, but instead "pulls" data from three development tools: the CVS configuration management system, the GNATS issue tracking system, and the MailMan mail archiving system. Applications of EPM include analyzing the similarity and diversity of software projects, code clone detection, and comparative analysis of open source project repositories.

Sixth Sense Analytics. This company [4] provides software measurement and analysis services based upon the use of sensors that send data to a centralized server, and even incorporates some Hackystat code into their system. Unlike Hackystat, users cannot download the server and store their data locally. Sensors are available for a variety of IDEs and configuration management systems. Analyses currently support two proxies for developer effort: Active Time and Flow Time.

ECG. The ElectroCodeoGram (ECG) project [13] is sponsored by the Software Engineering research group at the Free University of Berlin. ECG monitors developer activities in order to represent "micro-processes" during software development. Examples include the "copy-paste-change" micro-process, which is a common way of producing similar functionality in multiple locations in a software system, but which is has been hypothesized to produce defects more often than a "refactoring" micro-process, in which the common functionality is extracted out into a new method and called from both locations. ECG is intended to support empirical research into these and other micro-processes.

SUMS. The Standardized User Monitoring Suite (SUMS) project [12] is sponsored by the Pittsburgh Supercomputing Center and IBM. SUMS provides unobtrusive monitoring of developers, but accomplishes this not through individual sensors for specific tools, but rather through low-level operating system monitoring. SUMS has been used within a specially instrumented lab to collect data on student programmers in order to better understand the use of next generation high performance computing languages and tools.

All of the above ISEMA systems and Hackystat share a common feature: after installation and configuration of the system, data collection is unobtrusive and automatic. This is because the in-process metrics collected by these systems are inherently voluminous and thus impractical to collect manually. On the other hand, none of these other systems appears to have gone as far down the road to "genericity" as Hackystat. The next section begins our discussion of Hackystat requirement and design trade-offs by focusing on those we made early on in order to be "generic".

3. Primary trade-offs for ISEMA genericity

(1) Sensor-server architecture. An ISEMA system must perform two basic activities: data collection and data analysis, and almost always performs a third: data storage. There are a variety of top-level architectures one can choose to accomplish these goals, ranging from a single user approach where everything occurs on a single computer, to a client-server approach where data is collected on a client and sent to a central server for storage (either before or after analysis). Another architectural possibility is peer-to-peer, in which data is stored on individual computers but shared with others as required.

In Hackystat, we decided upon a client-server architecture in which the "clients" consisted of custom sensors developed for each tool to be monitored in the environment. The cost of this decision is the requirement that a custom software component must be created for a tool before its data can be included for analysis. The benefit is that the sensor can include domain knowledge about the tool whose behavior is being monitored. For example, the sensor for the Eclipse IDE can monitor the invocation of subsystems like the debugger and collect data about the subsystems being inspected, which can provide valuable insight into the process of development.

By convention, Hackystat sensors collect relatively "raw" data and send it to the server where all significant analysis occurs. This minimizes the processing overhead on the client computer. It also allows new analyses to be developed, deployed on the server, and then run retrospectively over previously collected sensor data.

Other ISEMA systems with different architectures illustrate some of the trade-offs inherent in this design decision. For example, SUMS does not require specialized sensors for each development tool, but instead instruments the operating system. This enables SUMS to transparently monitor any tool used by the developers without additional software development, though the type of data that can be collected by monitoring OS-level events is more limited than what can be obtained by custom software for each tool. Additionally, the SUMS instrumentation is specific to a single operating system.

EPM is another ISEMA system that does not use a sensor-server architecture. Instead, it "pulls" data from its tools using their public reporting interface. The trade-off in this case is the requirement that a tool have a reporting interface in order for it to be accessable to EPM using this technique.

(2) Workspaces. In an ISEMA system, measurements associated with files are collected automatically. A generic ISEMA system must confront the following problem: the same file could be named many different ways depending upon the operating system platform and use of configuration management. For example, the same file "Foo.java" might be associated with the file path c:\svn\projectA\Foo.java on developer A's computer and /usr/home/smith/svn-sandbox/projectA/Foo.java on developer B's computer.

In Hackystat, this issue is addressed by server-side post-processing of file names to create a canonical location known as a "Workspace", which has a canonical file path representation. The user must also provide a "Workspace Root" during configuration of their account, which enables the system to determine that directories rooted at c:\svn on Developer A's computer and /usr/home/smith/svn-sandbox/ on Developer B's computer might contain the same files.

An ISEMA system can avoid the need for the complexity of Workspaces in several ways. One approach is to simply disallow client-side collection of data about file artifacts, and instead collect this information from a single location, such as a configuration management system. Another way is to limit the ISEMA system to analysis of one user's data (i.e. not supporting aggregate analyses over groups of developers working on a common project), or requiring all programmers to use a common file system. EPM, ECG, and SUMS all avoid the need for workspaces through one or more of these simplifying assumptions.

(3) Projects. Most software engineers work on multiple tasks concurrently, and each task might involve a different set of collaborators. Many kinds of ISEMA analyses require the ability to organize the process and product data according to the task with which they are associated.

In Hackystat, this issue is addressed through a server-side abstraction called "Projects". When a Hackystat user defines a Project, she specifies a time interval, a set of Workspaces, and a set of email addresses corresponding to other Hackystat users. The server generates an email "inviting" the other users to join the Project. Accepting the invitation enables the system to perform project-level analyses that aggregate together the process and product data associated with each of the users. The specified set of Workspaces allows the system to filter out unrelated sensor data. This invitation system is required due to privacy issues, discussed further below.

One way to avoid the need for Projects is for the ISEMA system to guarantee that all data sent from a user is associated with a single task. This is the approach taken by the SUMS system, which is deployed in a laboratory setting under controlled conditions. Another way is to focus on analyses that are independent of particular tasks. For example, ECG identification of copy-paste-change micro-processes can be useful without associating their occurrence with specific tasks.

(4) Data Design and Quality Assurance. The requirement of regular, unobtrusive process and product data collection creates a number of challenges related to data design, completeness and correctness. For example, one cannot assume connectivity to the Hackystat server at all times: developers often work offline (such as when traveling), and the server can crash due to power outages or other problems. Second, sensors for different tools that perform the same type of function (for example, two configuration management tools such as CVS and SVN) should collect data in a standardized way and format so that analyses are not completely tool-specific. Finally, sensors can and will "drop out" occasionally due to power outages, platform changes, implementation bugs, and so forth.

In Hackystat, we provide a variety of mechanisms to address these data design and quality issues. First, a middleware application called the SensorShell provides infrastructure for Hackystat sensor development. The SensorShell provides a high-level API to sensor designers that insulates them from the low-level details of SOAP data transmission. It also transparently implements client-side offline data caching and re-transmission. Thus, if a developer is working on a plane or the server is unavailable, their data will be collected and cached on their laptop until she lands and re-establishes a server connection. Upon the next invocation of a sensor-enabled tool, the accumulated data will be sent to the server. The SensorShell also buffers sensor data locally and sends the collected data in a single SOAP request. By default, this results in sensor data transmission by a client once every ten minutes, which significantly lowers the overhead of sensor data transmission for the client and data reception by the server.

Second, to ensure consistent data collection across different tools, we developed the "Sensor Data Type" (SDT) abstraction. Among other things, sensor data types allow you to specify required fields indicating data that must be provided by all sensors supporting this SDT, as well as a "property list" field that supports an arbitrary amount of optional key-value data. For example, the "Commit" SDT includes required fields specifying data that all configuration management sensors must provide, but also allows a sensor for a specific tool to send additional optional data that may only be available for that particular tool type. The distinction between required and optional fields enables the development of "generic" analyses for Commit data that are independent of the specific configuration management tool, as well as specialized analyses for data that may be available on only one tool, such as Subversion.

Third, the requirement for unobtrusive data collection means that it is possible for one or more sensors to crash or otherwise stop sending data without any notification. On the other hand, sometimes sensor data is not sent simply because developers are not currently working with those tools. In Hackystat, we address this through "telemetry reports" that facilitate identification of missing sensor data. For example, on one occasion the report revealed that a developer was sending IDE, Build, and Commit data for a number of days without any corresponding Unit Test data. This anomaly helped the developer discover that his test sensor had become misconfigured during a recent upgrade.

One way to reduce the complexity of data quality assurance is to not use a sensor-based mechanism for data collection, and/or minimize the type of data that is collected. For example, the EPM project "pulls" data from three kinds of software engineering data repositories: CVS, GNATS, and MailMan. Simplifying the kinds of data that can be collected as well as the way in which they are collected can avoid some of the data quality issues we have needed to address in Hackystat.

(5) Configurability. A generic ISEMA system can be applied to many different measurement tasks, yet providing every single implemented analysis, sensor data type, and sensor in the system significantly impacts upon its usability. Java web application developers, for example, dislike wading through analyses focused on MPI (Message Passing Interface) programming using C++.

Clearly, a generic ISEMA system must be able to be tailored to the specific measurement and analysis concerns of an organization. In Hackystat, this is accomplished through two mechanisms. First, the build procedure allows an administrator to define a configuration that can both exclude public Hackystat modules implementing unnecessary functionality and include privately developed Hackystat modules implementing organization-specific functionality. Second, a set of extension points allow modules to implement generic functionality that can be extended for organization-specific purposes.

These configuration mechanisms have enabled us to grow the Hackystat framework to over seventy modules organized into four "subsystems". the "Core" subsystem provides essential functions that are independent of the specific sensors, sensor data types, and analyses contained in a configuration. Core functions include the sensor and sensor data type definition facilities, the SOAP data transmission capabilities, features for the web application interface, and the configuration mechanism itself. Most instantiations of the Hackystat framework include all of the core modules. The "Sensor" subsystem contains modules that each implement sensor support for a development tool. Each module in the "SDT" subsystem implements a sensor data type. Finally, the "App" subsystem contains modules that operate over sensors and sensor data types to provide higher level analyses that enable an organization to use the data for project monitoring, quality assurance, and decision-making.

Configuration using modules and (an extensible) extension point mechanism add significant complexity to Hackystat development, installation, documentation, and use. First, the Hackystat build process must represent and manage module dependencies. For example, a configuration that includes a sensor that generates data using the UnitTest SDT must be sure to include the module defining that sensor data type. Second, as Hackystat has grown to over seventy modules, 300,000 lines of code, and a half dozen public configurations, it has become impractical for developers to test each of their changes over the entire system, leading to the need for automated daily build and error reports. Third, configurations make the build process more complicated, and have required a substantial amount of documentation to be developed, which must also be configurable!

The complexity of Hackystat configurations comes from our desire to minimize constraints on the kind of tailoring that can be done. For example, the decision by Sixth Sense Analytics to not make their server available (much less extensible) can enable a more simple configuration process and mechanism.

4. Emergent trade-offs

We consider the sensor-server architecture, workspaces, projects, data quality assurance, and configurability to be "primary" trade-offs: design decisions that follow more-or-less directly from our goal to make Hackystat as generic an ISEMA system as possible. Interestingly, a number of additional trade-offs follow as a consequence of these primary trade-offs.

(6) Non-real time analysis. We have been contacted several times by researchers interested in using Hackystat for "real-time" software engineering measurement and analysis, in other words, in domains requiring feedback to the user within a second or two of a measurement event. As a simple example, "cyclomatic complexity" is a well-known measure of a method or function's complexity, and it is easy in Hackystat to provide a sensor for a tool such as NCSS that computes this metric. A real-time application of this measure might, for example, continuously monitor the cyclomatic complexity of a function as it is being written, and underline the function definition in red as soon as the complexity exceeds a certain threshold value.

Many such "real-time" systems for software engineering measurement and analysis can be envisioned, but Hackystat is not the appropriate infrastructure for their development. This trade-off results from our sensor-server architecture, the SensorShell middleware component to buffer data and enable offline collection, and the resulting implication that sensor data may appear on the server minutes, hours, or even days after it has been collected by the client.

This implication actually creates flexibility in the way sensors are implemented. For example, we currently implement our Subversion configuration management sensor as a timer-based system that runs once a day and collects all CM events from the previous day. Unlike a more real-time, "hook-based" design, a timer-based implementation does not require root-level privileges for its installation and use.

Our experiences suggest to us that the decision to support "real-time" vs. "non-real time" ISEMA is a fundamental trade-off. We believe that an ISEMA architecture will be significantly simpler and support its domain more effectively if it supports either real-time or non-real time applications but not both. Interestingly, we know of no systems that focus explicitly on generic support for real-time ISEMA applications.

(7) Sensor data type evolution. As a consequence of its goal of genericity, Hackystat does not presuppose what types of measurement data will be collected and how this data will be structured. However, to facilitate understanding and correct analysis of measurement data, Hackystat provides the sensor data type definition facility, which among other things differentiates between "required" and "optional" data. Hackystat also does not presuppose the specific tools from which sensor data will be collected, but does mandate that all tools send their sensor data as instances of one or more sensor data types.

The ability to define new sensors and sensor data types over time enables an incremental and exploratory approach to ISEMA system development. For example, one can implement a sensor data type to support a single kind of size counting tool, such as LOCC, and then add sensors for additional size counting tools such as CCCC, NCSS, and SCLC that use the same sensor data type.

The problem, of course, is that experience with a broader set of tools often reveals inadequacies in the original sensor data type definition. For example, it is quite common when first defining a sensor data type to build in assumptions about the nature of the data that turn out to be peculiar to the first tool you are instrumenting. These hidden assumptions only become apparent once sensors for additional tools of that type are under development.

Once a sensor data type definition is found to be inadequate and the decision is made to improve it, one must deal with the question of what to do with the sensor data already collected under the old definition. For the first few years of Hackystat's development, the system required you to throw away the data collected under the old definition if you wished to upgrade it. This is an expensive solution, and in several cases led us to simply live with a "bad" sensor data type definition simply because we did not want to lose access to the data we had already collected.

Hackystat now provides the ability to "evolve" sensor data type definitions to incorporate new insights about the most appropriate set of required and optional data. The evolution is implemented in terms of a distinguished method in the sensor data type definition class which "lazily" evolves older versions of the sensor data upon access. This approach enables both old data stored on the server to be upgraded to the new definition when retrieved for analysis, as well as data received from clients that are still using an old version of a sensor that has not been upgraded to use the new SDT definition.

Sensor data type evolution adds complexity to the representation and implementation of sensor data types, but this trade-off does enable a more exploratory style of development while preserving the benefits of typed data. To our knowledge, no other ISEMA system implements sensor data type evolution. If that is the case, then we hypothesize that other systems deal with this issue using one or more of the following trade-offs: (1) represent sensor data in an unstructured, non-typed format, (2) perform a thorough domain analysis prior to sensor data type definition in order to ensure that the structure is correct, or (3) force users to throw away old sensor data if the sensor data structure is changed.

(8) Intermediate abstractions. The design decisions to send sensor data in its "raw" form, combined with the Project abstraction for representing group activities on subsets of sensor data have led to the need for a variety of intermediate abstractions to support server-side analyses in order to provide acceptable performance and avoid redundant computation.

To see why this is so, consider two simple measures: an integer indicating the total time in minutes spent editing Project-related files by all members for a given day, and an integer indicating the change in size (LOC) of the system under development for this Project during that same day. There are two interesting things to note about the computation of these two integers. First, they are not particularly domain-specific measures: many of the Hackystat application domains find them to be useful, from software project telemetry to high performance computing to NASA's Mission Data System. Second, computing these two integers requires retrieving all of the sensor data sent by all Project members for the week of interest, filtering out the sensor data sent by members not related to this Project, and processing the remaining data appropriately. In Hackystat, several thousand sensor data points might require processing to compute each of these measures, and recurrent analyses like software project telemetry might use these measures for several weeks or months at a time on a regular basis.

Thus, an emergent trade-off in Hackystat is the use of cached, intermediate abstractions that represent "partial analyses" of the raw sensor data. Examples of these abstractions include "DailyAnalysis", which represents individual developer activities for a given day in five minute chunks, "DailyProjectData", which represents one or more measures for a given Project and day, "Reduction Functions", which compute sequences of measures over a given time period, and "SDSA Episodes", which partition a stream of developer behavioral events in discrete episodes suitable for later classification. Although these abstractions significantly improve performance, the trade-off is the cost and complexity of designing and implementing thread-safe caches with appropriate expiration policies which could be wholly eliminated by rebuilding the abstractions from scratch upon each request and using built-in database transaction mechanisms to manage concurrent access.

5. Scalability trade-offs

A third category of requirement and design trade-offs in Hackystat relates to "scalability." Scalability trade-offs address the total number of users that can be supported by a running server, of course, but also includes scalability with respect to analyses and public accessability.

(9) Usage scalability. The most obvious scalability trade-off involves the total number of users who can access a running system with acceptable responsiveness. Hackystat's server runs within Tomcat, and our public server currently accomodates several hundred users on a low-end Dell server with 2 GB RAM and 80 GB disk space.

Assessing usage scalability in an ISEMA system occurs along two primary dimensions that can be assessed independently. The first dimension is scalability with respect to data collection: in other words, how many concurrent users can be sending data to a server with acceptable responsiveness? Along this dimension, the trade-off in Hackystat to support only non-real time analysis, which enables client-side buffering, enables Hackystat to scale quite well with respect to data collection. Given that a single user will transmit data to a Hackystat server approximately every 10 minutes, and that a typical sensor data transmission involves only a few thousand bytes of data, it is easy to see that a single Hackystat server can provide adequate responsiveness to hundreds of concurrent users with a low-end server hardware configuration.

The second dimension is responsiveness with respect to user-initiated analyses. This assessment is of course entirely dependent on the specifics of the analysis, but we can offer one interesting insight from our experience with Hackystat. When we began development of Hackystat in 2001, we decided to store sensor data using an XML-based flat file organization, where a separate file would be used to store the sensor data of a given type for a given user on a given day. We viewed this as a "spike" solution that would simplify installation and debugging in the short-term, but would be replaced with a more robust and efficient RDBMS such as MySQL or Derby when system performance became constrained by this design decision.

To our surprise, after a small amount of performance optimization early on, our profiling activities have never since revealed our XML storage approach to be the bottleneck in responsiveness to user-initiated analyses. We believe this is due to Hackystat's use of cached intermediate abstractions, which significantly reduce the amount of raw sensor data access. Instead, our performance optimization efforts have focussed on this level, and have involved tuning the cache mechanism to avoid excessive heap usage as well as improvements to the intermediate abstraction implementations, such as pre-computing frequently used analysis results to minimize redundent sensor data access.

(10) Analysis scalability. Another scalability trade-off involves the ability of users to customize specific analyses. For example, the software project telemetry application allows users to monitor sets of measurement trends over time and looking for co-variances that suggest causal dependencies. For example, if one notices that code coverage is decreasing over time and that the build failure rate is increasing, it suggests that these two measures might be interrelated and that one could potentially decrease the build failure rate by improving test quality. A usability problem with software project telemetry is that the number of possible measures and co-variances increases exponentially with the number of available sensor data streams, so that even a half dozen sensor data streams leads to many thousands of potential telemetry reports.

Another example of this analysis scalability problem in Hackystat is the Software Development Stream Analysis (SDSA) application, which supports workflow analysis over sequences of developer behavioral events, such as opening a file, invoking a test case, refactoring code, and so forth. Once again, the range of approaches to partitioning the sequence of events into episodes and then classifying them as workflow states is very large.

In both cases, analysis scalability was improved through the implementation of a domain-specific language. The DSL for software project telemetry was implemented using a grammar, the JavaCC parser generator, and a custom interpretor. The DSL for SDSA was implemented using JESS and a set of CLISP rules.

(11) Data access scalability. One of the most important and complicated trade-offs in scalability for ISEMA systems involves the degree and form in which data collected from a system and its developers is made available to others. For example, in Hackystat, a measure called "Active Time" represents the number of minutes that a developer spent actively editing files associated with a project during a given time interval. A few developers have told us that they would never allow Hackystat in their organization because of the potential for this kind of data to be collected and the potential misuse of this data by management. Their (legitimate) worry is that measuring just the time spent editing files could be misinterpreted by management as a meaningful measurement of individual developer productivity, resulting in counterproductive measurement dysfunction within the organization. In these situations, we recommend that an organization adopting Hackystat begin by collecting only product measures and do not install sensors into their developer's IDEs until the developers become comfortable with that form of measurement. Systems like SUMS, which collect OS-level data at the level of individual keystrokes, have even more potential for user concerns regarding data access.

At one end of the spectrum, an ISEMA system could declare that all data is completely private and restricted to the user that collected it. While this "solves" the privacy issue arising from data access, it also severely impacts on the potential utility of the system. For example, it would not be possible to generate analyses that represent aggregate time, defects, churn, and so forth for a group of developers working on a single project.

Hackystat currently takes a more moderate approach to data access scalability, in which users can participate in Projects which enable their raw sensor data to be accessed for project-level analysis, although participation in a Project does not allow members to access each others raw sensor data directly.

Data access scalability beyond the project-level could provide very useful software engineering insights. For example, when paired with demographic information about the project and its developers, ISEMA systems could become a rich source of information about development issues and approaches at the language or application level. For example, the current Hackystat repository contains rich data about Java-related software development over the past five years, and we have already performed preliminary design work for "federated" Hackystat servers that can share and exchange qualitative and quantitative data [9], yet implementation awaits the definition of suitable policy for privacy protection.

(12) Developer community scalability. Our final trade-off concerns the potential for community involvement in the enhancement and customization of an ISEMA system. At one end, an ISEMA system can be developed as a completely closed source system, with no possibility for end-user extension. Such an approach minimizes the cost of developing the resources and mechanisms to support a broader developer community, and also enables all information about the design of the system to be kept proprietary.

Hackystat has chosen the opposite end of the spectrum, in which the source is freely available, the system is modularized for third party extension, mailing lists are provided for developers and users, extensive developer documentation has been developed, and a significant number of customizations to our build (Ant), test (Junit), and documentation framework (DocBook) have been developed to support community involvement.

Other ISEMA systems make different trade-offs. For example, Sixth Sense Software keeps the server proprietary, but allows end-user involvement in sensor development.

6. Buy vs. Build

A practical application of these trade-offs is to support the buy vs. build decision by an organization that wishes to introduce an ISEMA system.

First, for organizations that wish to avoid the investment in developing an ISEMA system in-house, these trade-offs can help you to evaluate the features and capabilities of various systems. For example, if you work in a multi-platform, collaborative setting, then whatever ISEMA system you choose should address the issues that were solved in Hackystat by the Workspace and Project abstractions.

Second, if you are considering developing your own ISEMA system, these trade-offs reveal the requirement and decision complexities that can result from the desire to provide genericity with respect to the types of data collected, the way in which data is collected, and the way in which data is processed. In some cases, a significantly simpler ISEMA solution is possible by eliminating one or more dimensions of genericity. For example, if the only development tool you use is Eclipse, and the only platform you work on is Windows, then much of the complexity of Hackystat can be avoided.

Third, regardless of whether you are buying or building, you must actively consider the issue of data privacy. An ISEMA system that can monitor developer behavior, such as the IDE sensors for Hackystat, has the potential to be viewed as "Big Brother" by developers and arouse fears that the ISEMA data will be used to evaluate developer productivity. Such counter-productive applications have been termed "measurement dysfunction", and for a more comprehensive treatment of this subject, we recommend that you start with Robert Austin's excellent overview [1].

7. A Research Agenda for ISEMA

When we began the Hackystat Project five years ago, we were not aware of other active ISEMA development projects. We are gratified that a community of users and developers interested in in-process software engineering measurement and analysis has arisen, and that Hackystat is in the vanguard of these efforts.

To conclude this paper, we wish to reflect on the implications of these requirement and design trade-offs for the software engineering research community in general and the ISEMA research community in particular. What are important issues to be addressed by ISEMA systems during the next five years? Here are six promising directions for future research and development.

Real-time ISEMA. Apart from ECG, a notable gap in current ISEMA research and development is the lack of support for real-time responsiveness, particularly in a distributed, collaborative setting. Infrastructure for real-time ISEMA could take at least two forms: real-time ISEMA feedback, and real-time ISEMA notification. Similar to the way in which spelling checkers went from a stand-alone, batch mode to a real-time "squiggly underline" mode, real-time ISEMA feedback could enable developers to monitor the measurement impact of their system after each keystroke. Real-time ISEMA notification is a form of remote feedback–instead of getting immediate ISEMA feedback on your own keystrokes, you get real-time immediate feedback on your fellow developer's keystrokes. Many interesting research issues exist in this domain, from the kinds of measures that would be generally useful in real-time, to the contextual information required to avoid "false positive" notifications.

ISEMA interoperability. As noted above, one result of our research is the insight that there can be no "one-size-fits-all" ISEMA system for the same reasons that there can be no "one-size-fits-all" programming language: the same design decisions that make Ruby an excellent language for small-scale web application development also make it inferior to Fortran for climate modeling. There will be a community of ISEMA systems that satisfy different organizational contexts and measurement goals.

A community of ISEMA systems provides the opportunity to establish standards for data representation and interoperation that would create new measurement and analysis alternatives for organizations. First, data collected in one environment could be analyzed using another environment. Second, standards for interoperability could enable "meta analysis", in which common and comparable features from a set of analyses from different environments are extracted and used to form more general conclusions. Finally, the process of standard setting could help facilitate propogation of experience across the ISEMA development landscape, increasing the rate of progress and bring stability to measurement and analysis definitions. Some preliminary work in this area has been done by the ECG developers, who provided partial interoperability between ECG and Hackystat.

Support for qualitative data. ISEMA systems currently focus on collecting and analyzing numeric data. However, rich insight into software engineering practices can be obtained through non-numeric, qualitative approaches such as grounded theory [14]. In our research on the software engineering of high performance computing systems, we have already discovered the need to integrate qualitative data (from developer interviews, journals, and researcher analyses) with quantitative process and product data. Yet no ISEMA system provides facilities for qualitative data collection and analysis comparable to what they provide for quantitative data collection and analysis.

Improved data privacy, anonymity, and access. One of the most important research issues for ISEMA systems is to better understand and manage the inherent tension between data privacy, anonymity, and access. As discussed above, guaranteeing complete privacy hinders even simple forms of analysis, such as the aggregate impact of group activities on a project. However, even the *potential* availability of certain types of measures to management can become a barrier to data collection itself, and/or lead to measurement dysfunction. On the third hand, providing broader access to the data collected by ISEMA systems has significant potential to provide new insights about software engineering.

Providing broader access to ISEMA data must somehow resolve the following conundrum: in order for ISEMA data to be interpreted correctly by the broader research community, contextual data about its collection and analysis must be available. On the other hand, such contextual data often reveals, either implicitly or explicitly, the identity of the organization, projects, and/or individuals from which the data was collected. This violates their privacy. Basili and others have began to work on this issue [2].

Automated decision making. Like other ISEMA systems, Hackystat collects data, and provides analyses based upon this data, but does not "act" upon these analyses. Instead, it is left to a developer or manager to review the data and decide what, if any, action to take based upon the analysis. As our experience and confidence in ISEMA systems grows, a natural next step is for the system to begin "closing the loop", by actually initiating development actions based upon its analyses.

As a simple example, large systems often develop test case suites that are so large and expensive to execute that they can be run in their entirety only intermittently. For example, the Mission Data System test suite was run only on weekends since it took over a day to execute and required machine resources not available during the working week. In such situations, developers must rely on partial testing before committing their code. An test case selection mechanism that includes ISEMA data could be integrated into a daily build mechanism and automatically decide which set of test cases to execute that would be most likely to reveal introduced defects while staying within the resource constraints of the environment.

Measurement and analysis validation. At the end of the day, collecting software engineering process and product measurements, and even inventing plausibly interesting analyses about this raw data, is not very hard. What is hard is validation: ensuring that (a) the collection mechanisms are actually collecting the data that they are supposed to be collecting and (b) the analyses performed on the collected data actually provide accurate and useful insight into the corresponding software development. As the ISEMA community matures, much more effort must be devoted to measurement validation.

Measurement validation almost always requires an independent source of information about the measure being validated. For example, we recently performed a pilot validation of the Zorro system for recognizing Test Driven Design [10]. Zorro is a Hackystat-based system that collects sensor data and analyzes it using a rule-based system that first partitions the stream developer's activities into "episodes", and then analyzes the episode sequence with the goal of determining conformance to test driven design practices.

While the system seemed reasonable in theory, we didn't really know if the sensors were collecting the right kind of data to support the right kind of episode partitioning, and if the resulting episodes were being classified correctly, and we couldn't use Zorro's data and analyses to validate itself. To perform the validation, we implemented a separate system, called the Eclipse Screen Recorder, that created a QuickTime movie of the Eclipse screen while the developer was working. By comparing this independent source of data about the developer's activities to the Zorro analyses, we were able to determine that Zorro was collecting the right data and analyzing it appropriately around 90% of the time.

References

[1] R. D. Austin. *Measuring and Managing Performance in Organizations*. Dorset House Publishing, 1996.

[2] V. Basili, M. Zelkowitz, D. Sjoberg, P. Johnson, and T. Cowling. Protocols in the use of experimental software engineering artifacts. Technical report, Submitted to Empirical Software Engineering, 2006.

[3] B. Boehm, C. Abts, A. W. Brown, S. Chulani, B. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece. *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000.

[4] G. Burnell. Sixth Sense Analytics Home Page. http://www.6thsenseanalytics.com/.

[5] 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.

[6] P. M. Johnson, H. Kou, J. M. Agustin, Q. Zhang, A. Kagawa, and T. Yamashita. Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, Los Angeles, California, August 2004.

[7] 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.

[8] 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.

[9] P. M. Johnson, B. T. Pentland, V. R. Basili, and M. S. Feldman. Cedar – cyberinfrastructure for empirical data analysis and reuse. Technical Report CSDL-05-02, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, May 2005.

[10] 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.

[11] 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.

[12] N. Nystrom. Standardized User Monitoring Suite (SUMS) Home Page. http://productivity.psc.edu/.

[13] F. Schlesinger. ElectroCodeoGram (ECG) Home Page. http://www.inf.fu-berlin.de/ecg.

[14] C. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4), July 1999.

[15] G. Succi. Professional Metrics (PROM) Home Page. http://www.prom.case.unibz.it/.

[16] K. Torii. Empirical Project Monitor (EPM) Home Page. http://www.empirical.jp/English/index.html.