Supporting development of highly dependable software through continuous, automated, in-process, and individualized software measurement validation

Philip M. Johnson
Collaborative Software Development Laboratory
Department of Information & Computer Sciences
University of Hawaii
johnson@hawaii.edu / 808-956-3489

July 2, 2002

Contents

1 Overview

1.1 Motivation

Highly dependable software is, by nature, predictable. With respect to behavior, one should know with confidence the situations under which the software will behave correctly and the situations under which it will fail. With respect to development, one should know with confidence how to assess the dependability of existing components and how to improve the dependability of components that are not sufficiently dependable. Finally, with respect to planning, one should know with confidence the cost and resources required to obtain the desired level of dependability. Unpredictable systems create dependability risks with respect to behavior, development, and planning, leading to dependability failures involving availability, reliability, safety, confidentiality, integrity, and maintainability [3].

Empirically-based approaches to creating predictable software are based on two assumptions: (1) historical data can be used to develop and calibrate models that generate empirical predictions, and (2) there exists relationships between *internal* attributes of the software (i.e. immediately measurable process and product attributes such as size, effort, defects, complexity, and so forth) and *external* attributes of the software (i.e. abstract and/or non-immediately measurable attributes, such as "quality", the risk of a specific component's failure in the field, and so forth). *Software measurement validation* is the process of determining a predictive relationship between available internal attributes and correspondingly useful external attributes and the conditions under which this relationship holds [18]. A validated software measure thus predicts things of interest about the future of a software development project and its resulting products based upon information at hand. Measurement validation is particularly important for quality: the ISO/IEC international standard on software product quality states that "Internal metrics are of little value unless there is evidence that they are related to external quality" [50].

Many recent attempts to create valid software measures focus on prediction of a specific external measure: the fault-proneness of individual modules after release. This focus is due to findings that the distribution of faults across modules tends to be non-uniform [49, 22, 47]. For example, 47 percent of the faults found by users of OS/370 were associated with only four percent of the modules. A valid measure for fault-proneness identifies a subset of modules at higher risk for post-release faults, and can be used to achieve more efficient and effective deployment of quality assurance resources. Predictable module-level fault-proneness is one of many possible ways to reduce dependability risks through valid software measures.

Recent research indicates the feasibility of creating valid software measures in both academic settings [4, 17, 51, 13, 15] and industrial settings [11, 6, 44, 27]. Despite these encouraging findings, broad application of valid measures is hampered by a number of problems described in Section 3.2, including lack of empirical evaluation in production settings, the possibility of confounding variables that reduce the predictive value or generality of the measure, the unknown impact of defect contagion and inheritance structures, and the overall cost of measure validation [5, 18, 8, 22].

This research project includes technological and methodological innovations designed to address these issues and advance the state of the art in both highly dependable computing and software measurement validation. It will do so through a set of measurement validation projects involving the Mission Data System at Jet Propulsion Laboratory and a Testbed adaptation of that software architecture.

1.2 Objectives

The general objective of this research is to design, implement, and validate software measures within a development infrastructure that supports the development of highly dependable software systems. The measures and infrastructure are designed to support dependable software development in two ways: (1) They will support identification of modules at risk for being fault-prone, enabling more efficient and effective allocation of quality assurance resources, and (2) They will support incremental software development through continuous monitoring, notifications, and analyses. Empirical assessment of these methods and measures will advance the theory and practice of dependable computing and software measurement validation and provide new insight into the technological and methodological problems associated with the current state of the art.

We will pursue this general objective through the following seven specific objectives:

(1) We will enhance the Hackystat system [28] with mechanisms for unobtrusive, continuous, and automatic collection of five classes of internal product and process measures of the Mission Data System (MDS) architecture for reusable flight control software developed by Jet Propulsion Laboratory (JPL);

(2) We will build risk models for predicting fault-proneness in MDS modules based upon current best practices [4, 11, 17, 9];

(3) We will use the in-process alert mechanisms of Hackystat to integrate these risk models into ongoing development, so that developers obtain "just-in-time" notification of modules at risk for pre-release or post-release faults.

(4) We will replicate objectives (1)-(3) on the MDS HDCP Testbed software, and compare the risk models obtained for the MDS architecture to the testbed;

(5) We will perform quantitative evaluations of the accuracy of the models and qualitative evaluations of the usefulness of the alert-based integration for ongoing development;

(6) We will make the results broadly available by publishing the open source Hackystat software (comprising sensors, risk models, and alerts) and by publishing the model definitions and our evaluation results in peer-reviewed journal articles. (The actual process and product data will remain proprietary to JPL). Our successes and failures will facilitate future advances in dependable computing and software measurement validation.

(7) We will enhance the infrastructure for education by developing software engineering curriculum modules that enable academic and industrial practitioners to learn how to use Hackystat to support software measurement validation for highly dependable computing in a classroom or industrial setting. The open source Hackystat software will enhance the infrastructure for research by facilitating replication and enhancement of this research.

2 Results from prior NSF-sponsored research

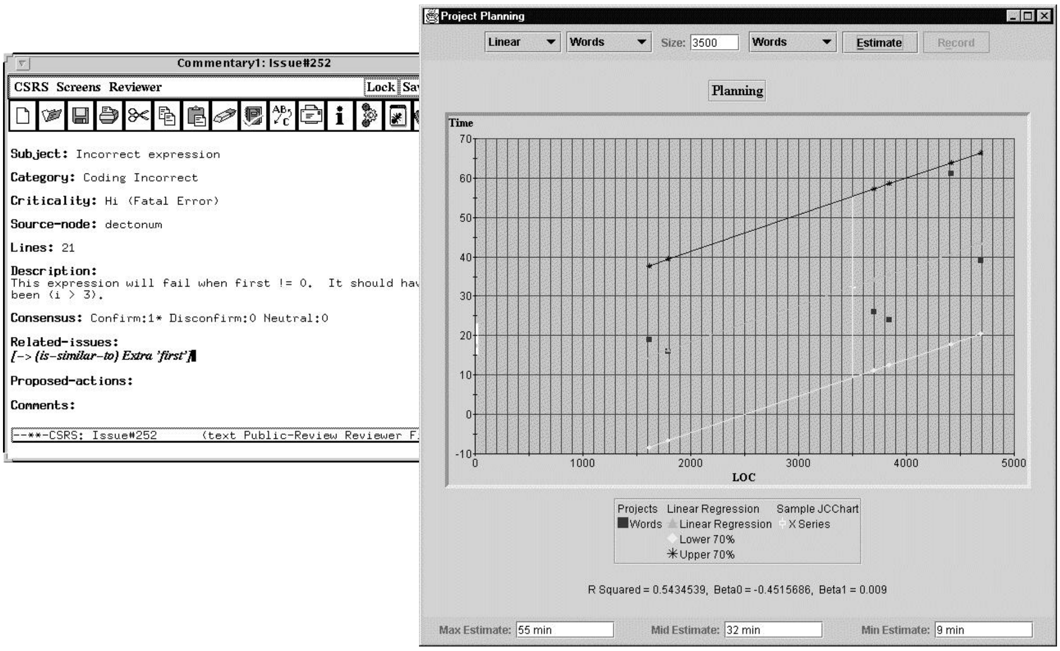
Our proposed research on software measurement validation through incremental, automated, in-process, and individualized software measurement validation follows directly from the findings of our last two NSF-sponsored research projects.

2.1 The CSRS project

Award number:	CCR-9403475
Program:	Software Engineering
Amount:	$161,754
Period of support:	June 1995 to June 1998
Title of Project:	Improving Software Quality through Instrumented Formal Technical Review
Principal Investigator:	Philip M. Johnson
Publications:	[32, 33, 39, 53, 40, 30, 31, 38]

In this research, we designed, implemented, and evaluated a collaborative software review system called CSRS and performed a series of experiments. Contributions of this research include the following:

Design of instrumented review technology. The CSRS system illuminates the design and architectural issues involved with the creation of a review system with both fine-grained measurement support and a process



Subject: Incorrect expression

Category: Coding Incorrect

Criticality: Hi (Fatal Error)

Source-node: dectonum

Lines: 21

Description:
This expression will fail when first != 0. It should hav
been (i > 3).

Consensus: Confirm:1* Disconfirm:0 Neutral:0

Related-issues:
[-> (is-similar-to) Extra 'first']

Proposed-actions:

Comments:

```
--**-CSRS: Issue#252      (text Public-Review Reviewer F
```

Size: 3500

R Squared = 0.5434539, Beta0 = -0.4515686, Beta1 = 0.009

Max Estimate: 55 min   Mid Estimate: 32 min   Min Estimate: 9 min

Figure 1: *Screen shots illustrating the Emacs-based CSRS interface for issue reporting (left) and the Java-based Leap interface for project size and time estimation (right).*

modeling language able to express a wide variety of current software review techniques. For example, in addition to traditional review metrics such as the number of issues discovered during different review phases, CSRS allows one to measure the time spent on review of any single function in the source code, and the sequence of functions visited by a reviewer over the course of review. The CSRS process modeling language allows emulation of most review techniques such as Fagan inspection [21] and Active Design Reviews [52], as well as new review techniques such as FTArm that require an online environment.

Instrumentation support for experimentation. The measurement support and process flexibility provided by CSRS also provides excellent infrastructure for controlled experimentation. We used it to investigate the costs and benefits of meetings in software review by implementing two review techniques, one which involved a synchronous meeting between review team participants to discuss issues, and another in which all team participant interactions were asynchronous. Our study found that the meeting-based method was significantly more costly in terms of effort than the non-meeting-based method, though we could not detect any difference in the numbers or types of defects found.

Comparison with Maryland experimental data. To gain insight into the generality of our findings on the efficiency and effectiveness of software review meetings, we teamed with Professor Adam Porter of the University of Maryland to perform a modified form of meta-analysis we call ”reconciliation” on the data resulting from the CSRS-based experiment and an independently designed and conceived experiment performed by Porter at the University of Maryland. Our reconciliation process differs from traditional meta-analysis in that we did not attempt to combine the data sets; instead, we formulated a set of common hypotheses and tested them against each data set independently. Despite differences in the types of documents reviewed, the backgrounds of the participants, and the technological infrastructure support between the two experiments, the two studies confirmed each other’s results for all six common hypotheses, providing further evidence that meeting-based review may not be the most efficient and effective choice for many development contexts.

2.2 The Leap project

Award number:	CCR-9804010
Program:	Software Engineering
Amount:	$260,000
Period of support:	July 1998 to December 2002
Title of Project:	Project LEAP: Lightweight, Empirical, Anti-measurement Dysfunction, and Portable Software Developer Improvement
Principal Investigator:	Philip M. Johnson
Publications:	[37, 36, 35, 34, 48, 16]

In this research, we built upon the results of the CSRS project as well as upon a case study of data quality in the Personal Software Process (PSP). In the case study, we subjected the data collected manually by students during the standard semester-long introduction to the PSP to a retrospective analysis of its accuracy. We implemented a database system to cross-check all calculations and and expose six classes of data defects. The analysis discovered over 1500 errors made by the 10 students, errors that in several cases lead to incorrect inferences concerning process improvement.

Based upon the usability problems we found with CSRS and the data quality problems we found with the PSP, we designed, implemented, and evaluated a Java-based toolkit called Leap. Leap satisfies four principal requirements. First, it is *lightweight*: it imposes a minimum of process constraints upon the user, it should be easy to learn and integrate with other methods and tools, and require minimal investment and commitment from management. Second, it is *empirical*, providing high quality collection and analysis of quantitative data. Third, it reduces the risk of *measurement dysfunction* [2], or the situation in which developers consciously or unconsciously skew data collection and analysis for personal, political or organizational goals. Finally, the toolkit and associated data is *portable*, allowing the user to continue to build their personal repository of software engineering data as they move within and across development organizations.

Contributions from the Leap project include the following:

Technology support for developer-centric, in-process, disruptive software project data collection and analysis. The Leap toolkit has been implemented and evaluated both through classroom and industrial use. The toolkit consists of approximately 40,000 lines of Java code and runs on all major platforms. The tool supports the collection of time, size, defect, and design pattern data, as well as the definition of size types, defect types, and projects and their associated components. The design of the Leap toolkit eliminates broad classes of data quality problems that users experience in the manual PSP.

Case study results on project estimation. The project estimation component of Leap provides 13 different possible estimation methods, in contrast to the PSP, which implements a single estimation method called PROBE. We performed a case study using the Leap toolkit in which we were able to retrospectively assess the estimation accuracy of the 13 methods. There were two surprising results. First, the PROBE method was sixth out of the 13 with respect to data accuracy. Second, the most accurate estimation technique was a form of "guesstimation", in which the developers, aided by the empirical data gathered by the tool and the results of the various estimation techniques, simply entered their best estimate.

Our successes and failures with CSRS and Leap deeply influence the design of the current research project, and give us an understanding of the variety of requirements that must be simultaneously addressed in order for a measurement-based system to achieve widespread, long-term adoption by developers across diverse organizations. First, the system's interface must be very simple and require little initial training in order to derive non-trivial benefits. Second, those who contribute the data must be those who derive benefit from the data, and those benefits must occur in the near-term, not the long-term. Third, empirical software project data is highly susceptible to measurement dysfunction, and mechanisms must be available to provide developers with the confidence that the data will not be used against them in the future. Fourth, and perhaps most importantly, even when developers experience substantial benefits from daily collection and regular analysis of empirical project data under mandated conditions, most begin “forgetting” to collect and analyze their data as soon as they are left on their own. In the case of PSP, we conjectured that this was due to the high overhead of manual daily collection and analysis. However, our experiences with Leap tool support indicate to us that the fundamental adoption problem of developer overhead will not be solved through normal, evolutionary approaches to measurement automation, such as GUI-based utilities to simplify daily entry of time, size, and defect data and associated analyses for project planning or post-mortems.

3 Related work

3.1 Advances in software measurement validation

Recent research indicates that valid software measures of fault-proneness may be both feasible and effective, particularly for object-oriented languages such as C++ and Java. Most recent research uses some variant of the Chidamber-Kemerer (CK) object oriented metrics [14] for at least a subset of the collected internal product metrics.

An empirical study of student-developed C++ programs showed that five out of six of the CK metrics appeared to be useful in predicting class fault-proneness [4]. Furthermore, these metrics appeared to be better predictors than traditional code-level measures (such as maximum statement level nesting, number of functions, etc.) A replication of this study with data from an industrial system developed by experienced, professional developers showed similar results, with import coupling being an especially strong and stable predictor of fault-proneness [11].

A case study of four systems showed that a custom “Coupling Dependency Metric” was the best predictor of run-time failures and residual faults [6]. The systems included a university course registration system with 80,000 lines of COBOL, a text formatting system with 3,000 lines of C, a patient management system with 82,000 lines of C++, and an electronic file transfer system with 6,000 lines of Java.

Bell Canada developed methods and tools that combine process capability assessment with source code metrics to support risk assessment during product acquisition [44]. Bell Canada obtains the source code for products it is evaluating for acquisition and uses the Datrix static analyzer to obtain metrics such as the number of declarative statements in each function, the ratio of control structure inside loops vs. outside loops, and the number of logical comments in a group. The values obtained are compared to predefined thresholds and used to identify weak components, which may be subject to human inspection. Over 20 products and 100 million lines of source code have been assessed in this manner so far, though no controlled studies of its effectiveness have been published.

The Emerald and ARMOR systems used by Nortel Networks build models using a combination of product metrics (such as size and complexity) and process metrics (such as the number of changes since the last release and the number of developers who have edited this module since the last release) [27, 43]. In one study, data from a commercial product was used to develop two predictive models of fault-proneness, one using just product data and another using both product and process data. The model using both process and product data increased fault-proneness detection by approximately 20 percent [41]. The ComPARE System extends the ARMOR system with new metrics and quality models [12].

Although most studies use models based on single or multi-variable regression to relate internal to external metrics, one study combines Bayesian Belief Networks and fault trees to improve prediction by capturing causal dependencies between software engineering processes and software reliability [51]. Another study provides a comprehensive review of statistical modeling techniques for software measurement validation [17].

3.2 Limitations of software measurement validation

While the above research illustrates the promise of software measurement validation, these results and their application are limited in several ways.

First, the measures were typically validated by demonstrating the existence of a model that would fit selected internal measures (such as coupling) to selected external measures (such as fault-proneness) on an already completed project. In the academic studies, the critical next step of actually deploying the model on a new development project and evaluating its predictive accuracy was not taken. While Nortel and Bell Canada claim to actively use their models to predict risks and post-release quality, no controlled studies have been published that rigorously assess the accuracy and utility of those predictions.

Second, models built from either academic and industrial data may be subject to confounding variables. For example, since class size is known to be associated with the incidence of faults in object-oriented systems, models that do not control for size could be misleading with respect to their accuracy. This effect is illustrated by a case study in which two validation models were created from data on 174 C++ classes in a telecommunications system [5]. The first model does not control for size, and generates statistically significant relationships between four internal measures (the Chidamber-Kemerer metrics WMC, RFC, CBO, and LCOM) and an external measure (fault-proneness). In the second model, which controls for size, all of these statistically significant relationships between internal and external measures disappear. The likelihood of size as a confounding variable calls into question studies that do not control for size, such as those described in [7, 10, 24, 55].

Third, developer experience has also been cited as a confounding variable [17]. One would suspect that more experienced engineers would produce components with less faults. In this case, a negative relationship would exist between experience and faults. However, an organization might assign their most experienced developers to their most complex components. This would create a positive relationship between experience and the internal product measure (complexity). Finally, commonly used measures of developer experience (such as number of years of professional employment) might be too simplistic to capture the true variation in ability between developers.

Fourth, conflicting data exists involving defect contagion, or what should be predicted about post-release defects given a known pre-release defects. Positive contagion refers to the situation in which the discovery of pre-release defects in a component indicates that many more are likely to exist in the component and escape into the field. One source of evidence for positive contagion comes from an industrial study which found that components with more faults pre-release also tended to have more faults post-release [8]. On the other hand, negative contagion refers to the opposite situation in which the discovery of defects in a component prior to release reduces the likelihood of defects in that component in the field. Review methods such as Software Inspection are predicated upon negative contagion: the more thoroughly and rigorously the component is inspected and the more defects found, the less defects should remain in that component after release [23]. Thus, while both positive and negative contagion have been demonstrated to occur, it is not known how to predict which form of contagion exists in a given component at a given time, which makes it difficult to create a valid software measure for this kind of relationship.

Fifth, researchers have produced conflicting results regarding the relationships between internal and external measures. One study found that making changes to a C++ program with inheritance consumed more effort than a program without inheritance [13], while another study found that a C++ program with inheritance took less effort to modify than one without [15]. One study found a relationship between the depth of the inheritance tree and fault-proneness in Java programs [20], while another found no such effects with C++ programs [19]. Another study failed to find any predictive relationship between typical complexity measures and fault-proneness [22].

In summary, the state of art in software measurement validation indicates that internal measures show promise as a predictor of useful external measures (such as the risk of fault-proneness) which affect software dependability. However, no general consensus is emerging on what measures to validate and under what circumstances a measure that is validated in the context of one project with one set of developers in one application domain can be validly applied to another project with different developers and a different domain. Progress is also impeded by the high cost of manual collection and analysis of the data. Finally, most research is historical in nature and focussed on illustrating the potential applicability of valid software measures. Little has been done regarding the practical issues of how to make the measurement data available to ongoing software development in a timely and useful manner.

4 Current status

4.1 Overview of Hackystat

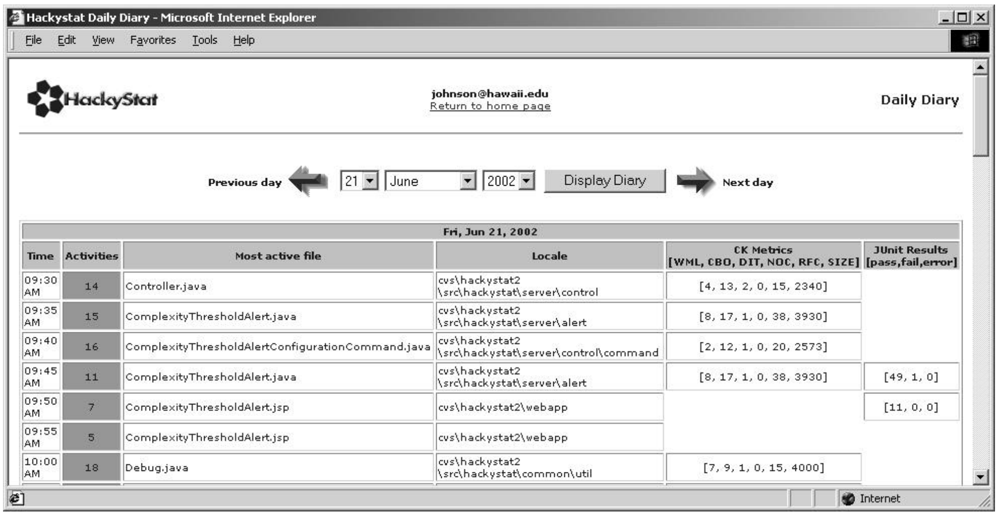
Hackystat is a client-server, sensor-based web service that provides a generic framework for unobtrusive collection and analysis of process and product measurements. The system was initially released in July, 2001, and is currently in active use at three sites. The design of Hackystat is influenced by our experiences developing instrumented applications for computer-supported cooperative work and software quality assurance over the past ten years [29, 32, 53, 36, 37]. We have learned that in-process, fine-grained measures have great potential for providing useful insight into work processes in general and measurement validation in particular, but that adoption is problematic unless the overhead of data collection and analysis is extremely low. This appears to be true regardless of the potential (long-term) benefits of the data.

Hackystat is designed to be used by developers in two contexts, which we call *measure-generation* mode and *measure-driven* mode. Among other things, measure-generation mode supports the discovery of valid software measures, while measure-driven mode makes the results of those measures available to developers in a "just-in-time" fashion.

To begin, assume that a valid measure is already available and the system is being used in measure-driven mode. (In reality, the system can be used in both modes simultaneously, with some valid measures used for analysis and prediction while others are being collected for subsequent exploration.) To start, developers download and attach custom built sensors to their development tools, such as their editors, unit testing tools, configuration management system, build tool, and so forth. These sensors unobtrusively monitor individual developer activities and send data on both process and product to a web server. Once a day, a variety of analysis mechanisms are invoked over the accumulated data. If these analyses discover "anomalies", the web service sends an alert via an email to the developer informing them that an anomaly has been discovered. The email also includes an URL to a page on the server that provides detailed empirical data about the nature of the anomaly. The developer can use this link to learn more about the problem, and also to provide feedback to the system on whether the alert was valid, i.e. that anomaly actually provides useful information to the developer.

The current implementation of Hackystat includes three kinds of sensors: one collects "activity" data on developer effort and focus of attention; another collects the current values of the Chidamber-Kemerer object oriented complexity metrics associated with the Java class file being worked on by the developer; and the third sensor collects pre-release defect data (through the results of running JUnit regression tests.) The "Daily Diary" page provides a summary of the data collected by each of these sensors for a specific developer during a single day, broken down into five minute intervals.

Figure 2 illustrates a portion of one such Daily Diary. For each five minute interval in which sensor data is recorded, the page displays the "most active file" that the developer interacted with, as well as its "locale" or region of the system (normally the directory structure with volume information discarded.) When the most active file is a Java file, the page displays five of the associated Chidamber-Kemerer metrics (Weighted Methods per Class, Coupling between Objects, Depth of Inheritance Tree, Number of Children, and Re-



Hackystat Daily Diary - Microsoft Internet Explorer
File Edit View Favorites Tools Help
HackyStat
Previous day
Display Diary
Next day
Internet

johnson@hawaii.edu
Return to home page

Daily Diary

21 June 2002

Fri, Jun 21, 2002					
Time	Activities	Most active file	Locale	CK Metrics [WML, CBO, DIT, NOC, RFC, SIZE]	JUnit Results [pass,fail,error]
09:30 AM	14	Controller.java	cvs\hackystat2 \src\hackystat\server\control	[4, 13, 2, 0, 15, 2340]	
09:35 AM	15	ComplexityThresholdAlert.java	cvs\hackystat2 \src\hackystat\server\alert	[8, 17, 1, 0, 38, 3930]	
09:40 AM	16	ComplexityThresholdAlertConfigurationCommand.java	cvs\hackystat2 \src\hackystat\server\control\command	[2, 12, 1, 0, 20, 2573]	
09:45 AM	11	ComplexityThresholdAlert.java	cvs\hackystat2 \src\hackystat\server\alert	[8, 17, 1, 0, 38, 3930]	[49, 1, 0]
09:50 AM	7	ComplexityThresholdAlert.jsp	cvs\hackystat2\webapp		[11, 0, 0]
09:55 AM	5	ComplexityThresholdAlert.jsp	cvs\hackystat2\webapp		
10:00 AM	18	Debug.java	cvs\hackystat2 \src\hackystat\common\util	[7, 9, 1, 0, 15, 4000]	

Figure 2: *Individual developer process and product metrics at five minute intervals. The activity sensor monitors developer actions, and infers the file that was the "most active" object of effort during each five minute interval. This file's directory structure provides the "locale", or region in the system where effort was occurring. When the most active file is a Java file, a second sensor calculates its Chidamber-Kemerer metrics from the .class file associated with the Java file, as well as its size in bytecodes. A third sensor monitors invocation of the JUnit regression testing tool and provides a summary of the results to this page.*

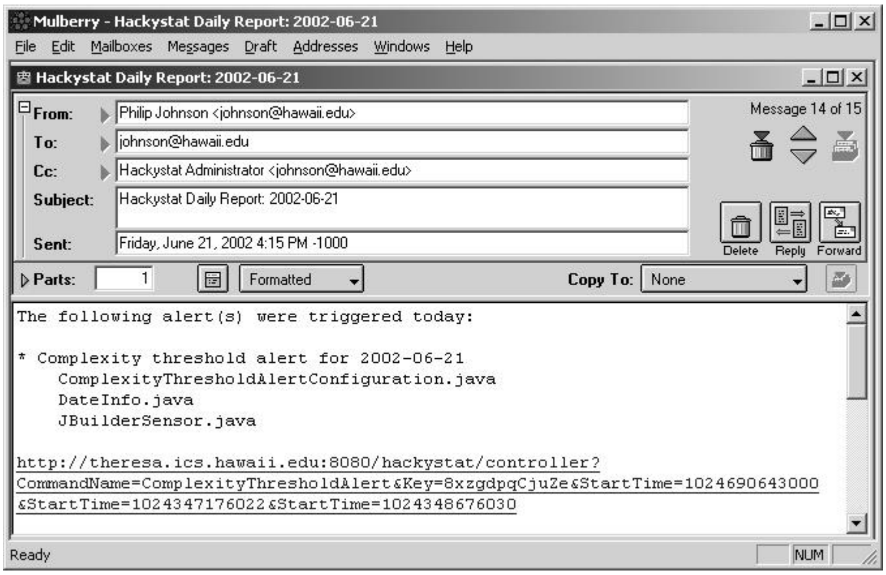
sponse for Class) as well as the size of the class [1]. Finally, the Daily Diary shows when pre-release unit tests are run and their results.

This Daily Diary page from the current implementation illustrates the ability of Hackystat to record and analyze both process measures (what file the developer is working on and when she is working on it; when are unit tests invoked) and product measures (what is the complexity of the class under development, how is it changing, and what is the quality of the current state of the system with respect to its performance during unit testing.) There are many ways to exploit these measures; for this example of measure-driven mode we will illustrate the use of the "Complexity Threshold Alert".

The complexity threshold alert monitors the values of the six complexity measures tracked for each Java class and sends the developer an email when threshold values for some combination of the measures exceeds predetermined values. Figure 3 illustrates the email that is sent to the developer when the system detects Java classes with complexity measures exceeding the thresholds. To obtain details on this anomaly, the developer retrieves the URL in the email, which brings up the page illustrated in Figure 4. If the alert algorithm was validated for fault-proneness, then these classes are candidates for additional quality assurance, such as Inspection, redesign/refactoring, or additional testing.

This example of automated collection, analysis, and feedback of complexity data, while simple, nevertheless supports a central claim of this research proposal: that Hackystat provides efficient and effective infrastructure for the development of valid software measures. In this case, the hypothesized valid measure consists of a useful, predictive relationship between a set of internal complexity measures and the external measure of post-release fault-proneness. The threshold value settings could be determined by adapting published valid measures [4, 11], through developer intuition, or through percentile rankings that identify the most complex classes relative to the system as a whole. Note that the current Hackystat sensors can support

[1] The size of a Java class is measured in bytecodes, which preliminary studies indicate to be very highly correlated (r^2 = .93) with non-comment source lines of code. On average, 40 bytecodes are required for each non-comment source line of code in a Java class.



Mulberry - Hackystat Daily Report: 2002-06-21
File Edit Mailboxes Messages Draft Addresses Windows Help
Hackystat Daily Report: 2002-06-21
Message 14 of 15
Delete
Reply
Forward
Ready
NUM

From: Philip Johnson <johnson@hawaii.edu>
To: johnson@hawaii.edu
Cc: Hackystat Administrator <johnson@hawaii.edu>
Subject: Hackystat Daily Report: 2002-06-21
Sent: Friday, June 21, 2002 4:15 PM -1000

Parts: 1 Formatted Copy To: None

```
The following alert(s) were triggered today:

* Complexity threshold alert for 2002-06-21
    ComplexityThresholdAlertConfiguration.java
    DateInfo.java
    JBuilderSensor.java

http://theresa.ics.hawaii.edu:8080/hackystat/controller?
CommandName=ComplexityThresholdAlert&Key=8xzgdpqCjuZe&StartTime=1024690643000
&StartTime=1024347176022&StartTime=1024348676030
```

Figure 3: *The system-generated email indicating a complexity alert. This alert was configured to check the most active Java files edited by this developer during the past week, assess their complexity, and send the developer an email if files are found that exceed certain threshold values. Clicking on the URL associated with this alert brings up the page illustrated in Figure 4.*

the development of a more sophisticated measure that augments the internal product measures of complexity with additional internal process measures including the effort expended on the class by the developer and the pre-release defect history.

Hackystat in measure-driven mode differs in four ways from prior approaches to software measurement validation:

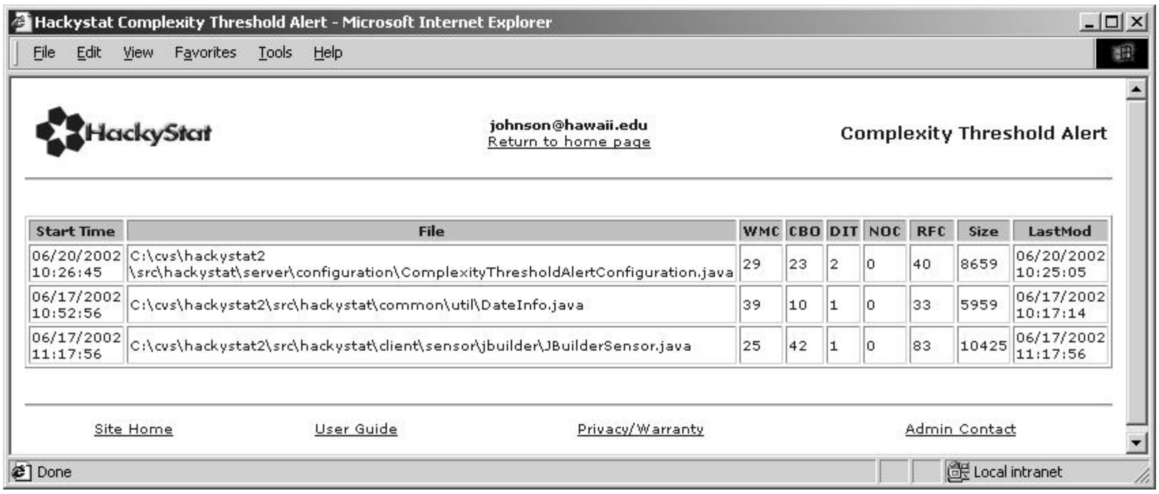
First, validation activities are *automatic*. Once the sensors are installed and the analysis mechanism activated, the developer does not have to manually collect or periodically “poll” the data analysis functions; the system will instead “interrupt” the developer when an event of interest occurs.

Second, validation measures can be *individualized*. Each developer’s data can be used to generate a personalized measure and/or alert scheme that reflects their activities and development style.

Third, measure application is *continuous*. In the above example, developer activities are monitored and product and process measures are reanalyzed each day as the developer changes old classes or implements new ones.

Fourth, the validation is *in-process*. Instead of gathering data from a set of developers on one project and hoping the measure applies to later projects with potentially different developers, Hackystat facilitates development of valid measures that feed back immediately into the current development process.

In contrast to measure-driven mode, in which one or more measures are assumed to be valid and thus useful for providing insight to developers on the status or trajectory of development, measure-generation mode is a more exploratory process in which the values of internal measures maintained by Hackystat are related to external measures (which may or may not be maintained by Hackystat.) The central objective of this research, which is to design, implement, and validate software measures that support the development of highly dependable software systems, will be pursued by defining and pursuing a methodology for measure-generation mode at Mission Data Systems and its subsequent application in measure-driven mode. This methodology will be described in Section 5; the next section introduces the Mission Data System development organization and development procedures necessary to motivate the methodology.



Start Time	File	WMC	CBO	DIT	NOC	RFC	Size	LastMod
06/20/2002 10:26:45	C:\cvs\hackystat2 \src\hackystat\server\configuration\ComplexityThresholdAlertConfiguration.java	29	23	2	0	40	8659	06/20/2002 10:25:05
06/17/2002 10:52:56	C:\cvs\hackystat2\src\hackystat\common\util\DateInfo.java	39	10	1	0	33	5959	06/17/2002 10:17:14
06/17/2002 11:17:56	C:\cvs\hackystat2\src\hackystat\client\sensor\jbuilder\JBuilderSensor.java	25	42	1	0	83	10425	06/17/2002 11:17:56

Figure 4: *Java classes exhibiting complexity above the threshold values. For this example, the threshold values were Weighted Methods per Class >= 20, Coupling between Objects >= 10, Response for Class >= 20, and Size >= 2000. If the alert implements a valid measure, then these files would be more fault-prone, and thus candidates for quality assurance such as Inspection, additional testing, or redesign/refactoring.*

4.2 Overview of the Mission Data System at Jet Propulsion Laboratory

Until recently, deep space missions tended to be one-of-a-kind, with distinct science objectives, instruments, and mission plans. For example, when Jet Propulsion Laboratory launched six missions in six months between October 1998 and March 1999, there was no common framework for developing mission software and little software reuse. The goal of the Mission Data System project is to develop a set of software architectures that accommodate the complexities of future mission requirements, including architectures for flight, ground, and test data systems. MDS facilitates software reuse, expands autonomous capabilities, and enables infusion of new software technologies. Users of MDS will benefit from a unified architectural framework for building end-to-end flight and ground software systems, along with executable example uses of those frameworks running a simulated mission [1]. MDS currently consists of approximately 5,700 C++ classes and 548,000 non-comment lines of code.

The MDS software has very high dependability requirements, and represents significant advances in the areas of software design, construction, and operation. Some of the architectural characteristics creating dependability challenges include: (1) the need for correct operation over a very large behavior space; (2) mission commands expressed as high level “goals” that are expanded into control sequences by on-board software; (3) architectural components that include reasoning components interacting with internal models of spacecraft structure and behavior.

The MDS development group employs a highly automated incremental development process with extensive tool support for configuration management, system build, and automated testing [46]. In general, each high-level development increment is organized into a set of work plans according to their impact on six areas: simulation, flight, transport, ground, framework, and test. Work plans are broken down into Implementation Tasks, which are further broken down into Change Packages. Change packages specify the actual files requiring modification to accomplish the enhancement and the responsible developer(s), and are the units of work input to the AllFusion Harvest Change Manager system [25]. Harvest maintains a workflow state model that tracks the progress of the Change Package as it goes through the states of “Dev Waiting”, “Dev”, “Dev Complete”, “Build Queue”, “Build Test”, “Integration Test”, “Test Complete”, and “Release”.

Dependability-related problems in the MDS development process can be manifested in many ways, but

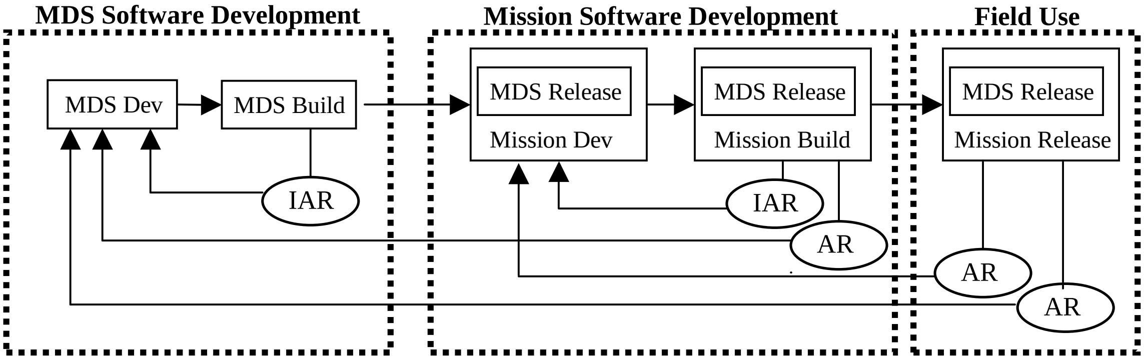


Figure 5: *MDS software development yields a release of MDS, which becomes a component during development of a specific mission's software. Internal Anomaly Reports (IARs) represent pre-release faults within a development process, while Anomaly Reports (ARs) represent post-release faults.*

three events are particularly important to this research project. First, a Change Package may fail during Build Test or Integration test. This normally indicates a pre-release fault, and is one of the situations leading the second event: the creation of an Internal Anomaly Report (IAR). Both build failures and IARs represent pre-release faults in the software. Once MDS is delivered to a customer and is under adaptation, use during either mission software development or field use may result in the discovery of additional faults. Each of these post-release faults will result in the third event: the creation of an Anomaly Report (AR).

Figure 5 provides an abstract view of the relationship between the MDS architecture development effort, the Mission software development projects that will utilize MDS as a component, and the ultimate field use of the mission software. The goal of the diagram is to point out how pre-release faults (represented by Internal Anomaly Reports) can be generated during MDS development or Mission software development, and that MDS post-release faults (represented by Anomaly Reports) can occur during either Mission software development or resulting field use. (Although the Mission software development process is not required to use MDS build technology and workflow representations such as IARs and ARs, they are employed here for simplicity's sake.)

The MDS project is an ideal testbed for the proposed research on highly dependable software. First, both the requirements for dependability and the costs of failure are extremely high, which means the organization is motivated and open to innovative approaches. Second, the current level of development process automation means that useful internal measures are readily available for collection by Hackystat sensors. Finally, the presence of two concurrent cycles of development (MDS and the Mission software adaptation) creates the possibility of combining internal measures from multiple project sources to generate new kinds of quality assurance models, as will be discussed next.

5 Research plan

5.1 Overview of valid software measure development for MDS

In the context of the MDS development process, "perfection" means satisfying customer requirements while simultaneously preventing all fault creation, as measured by the cessation of all Internal Anomaly Reports and Anomaly Reports. Fortunately, a perfect development process is not essential for very high dependability. Instead, high dependability can be pursued by improving the predictability of fault-proneness, as measured by improving the ability of the development organization (either MDS or a Mission-specific one) to predict characteristics of the future occurrence and nature of Internal Anomaly Reports and Anomaly Reports based upon the past history and current state of development.

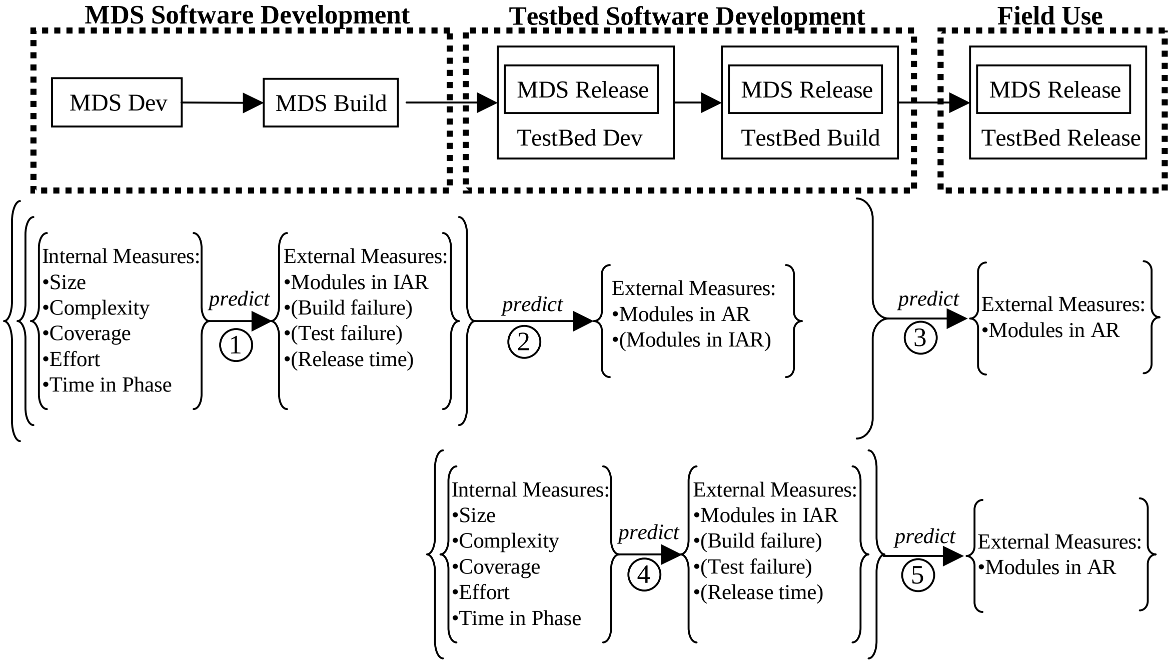


Figure 6: *The five measure validation experiments for the MDS and Testbed development projects. The top of the diagram shows that MDS development yields a release that is used as a component by the TestBed development process, which yields a release of a TestBed system that is deployed in the field. The middle of the diagram illustrates three measurement validation experiments using MDS process and product data to predict MDS, Testbed, and Field defects. The bottom of the diagram illustrates a replication within the Testbed development environment. (Prediction of external measures in parentheses will be investigated but are less important for dependability.)*

Figure 6 illustrates the proposed approach to improving predictability and thus dependability in the current MDS development context. It builds off the representation of faults as IARs and ARs as illustrated in Figure 5, and uses these in concert with internal measures for five software measurement validation experiments.

The top third of Figure 6 is similar to that in Figure 5, except the generic “Mission” software development in Figure 5 is replaced by an actual HDCP Testbed software development project [45]. The Testbed project is intended to simulate most aspects of an actual Mission development project, thus serving as a basis for dependability related research and to demonstrate the feasibility of MDS for actual mission development. Testbed development will result in a functioning software system that can be deployed and evaluated in field conditions.

While the diagram illustrates a linear process for simplicity’s sake, in reality the MDS and Testbed development processes are concurrent and interdependent: Testbed development may yield MDS ARs that lead to new releases of MDS that are incorporated into ongoing Testbed development. One research question to be explored is the impact of new releases on the predictive models and determining the circumstances under which recalibration becomes necessary.

5.2 Five software measurement validation experiments

The circled “1” in Figure 6 represents the first software measurement validation experiment. The left-most parenthesized list indicates a set of internal measures of MDS development (size, complexity, coverage, effort, and time in phase), while the adjacent parenthesized list indicates a set of external measures available later in development (the most important of which is the modules or files implicated in an Internal Anomaly Report, though build failure, test failure, and release time can also be analyzed). Using the methodology described in Section 5.3, we will attempt to develop a valid software measure that relates one or more of the MDS internal measures to the external measure of pre-release fault-proneness as evidenced by IARs. We will also explore the use of MDS internal measures to predict other external measures such as Build failure, Test failure, and the calendar time required for a Change Package to be released, although these are somewhat less important for dependability.

Recall from Figure 5 that while the MDS project can generate pre-release fault information, post-release fault information comes from a Mission-specific development process as well as from actual field use. The second software measurement validation experiment uses MDS Anomaly Report data from the Testbed development process. We will attempt to develop a valid software measure that uses both internal and external measures from within MDS development to predict the occurrence of MDS Anomaly Reports during Testbed development. Such a measure will help MDS developers identify in advance which modules are more likely to fail during adaptation to a specific mission context. This experiment is illustrated with the circled "2" in Figure 6 through the parenthesized lists of internal and external measures for MDS that predict two external measures of Testbed software development. (The parenthesized external measure, Modules in IAR, indicates that the analysis may yield a measure that helps identify fault-prone Testbed modules, although this is less important than identifying fault-prone MDS modules.)

The circled "3" in Figure 6 represents the third software measurement validation experiment. It combines data from MDS and Testbed development to predict those modules responsible for post-release defects during field use of the Testbed system.

The fourth and fifth software measurement validation experiments are replications of the first and second projects described above. In the fifth experiment, we will collect measures from Testbed software development and attempt to define a valid software measure that uses internal Testbed measures to predict pre-release module fault-proneness, as evidenced by IAR reports (or whatever representation is used in the Testbed development process). In the sixth experiment, we will use both internal and external Testbed measures to attempt to define a valid software measure for predicting module fault-proneness in the field. These last two experiments are represented by the circled "4" and "5" in Figure 6.

5.3 Experimental methodology

Each of the five validation experiments consists of four phases: Sensor Building, Baseline Data Collection, Model Construction, and Model Evaluation.

During the first phase, *Sensor Building*, we will implement any Hackystat sensors not yet available for collecting the measures of interest to the experiment. Sensors are specific to the tools and languages used by the developer group. For MDS, we will augment our current set of sensors for JUnit, Java, Emacs, and JBuilder with new sensors for C++ and the Harvest configuration management tool. The additional sensors required for the Testbed project will depend upon the development group selected for that project. Sensor building effort will be front-loaded: we expect to spend a significant amount of time during the first two years on Sensor Building, but relatively little time during the second two years.

During the second phase, *Baseline Data Collection*, we will use the deployed Hackystat sensors to begin automated collection of both the internal and external measures for one external release of the target system. Internal measures include the following product and process measures of a single module: the Chidamber-Kemerer metrics WMC, DIT, NOC, RFC, and CBO [14]; Briand's method invocation import coupling [11]; size (non-comment source lines of code); unit test coverage; developer effort; unit test failure rate; build failure rate; and interval time in build. External measures of a single module include: the number and severity of IARs related to this module; the number and severity of ARs related to this module. A module will be either a C++ class (in the case of MDS) or a Real-Time Java class (in the case of the Testbed project).

From an experimental point of view, the internal measures correspond to independent variables, and the external measures correspond to dependent variables.

During the third phase, *Model Construction*, we will analyze the baseline data collected for one or more external releases of the system and construct risk models that predict fault-proneness (i.e. at least one IAR or AR of a given severity level) for a module given the values of the internal measures (i.e. the product and process measures listed above). Our approach will follow best practice guidelines for validating software product metrics [17]. We plan to construct risk models based upon logistic regression which compute the expectation of a component being fault-prone given the values of selected internal measures (independent variables). Logistic regression is a standard classification technique based upon maximum likelihood estimation [26, 42]. To select the internal measures to be used in the risk models, we will use Baseline data to assess the strength of the relationship of each internal measure to the external measure after controlling for size. For models based upon logistic regression, the strength of the relationship is measured by the change in the odds ratio. To increase the stability of the model, we will seek a set of internal measures that are strongly correlated to the external measure while weakly correlated to each other, thus reducing collinearity. We will investigate the confounding variable of developer experience by building individualized models based on each developer's process and product data and comparing that to a model based upon an aggregate of all product and process data.

During the fourth phase, *Model Evaluation*, each model will be subjected to a goodness of fit analysis on the Baseline data. This analysis checks to see that the model has reasonable predictive accuracy for the historical data used to generate it. Following this retrospective evaluation, the model's usefulness will be evaluated in the context of the next external release cycle of development by being incorporated into Hackystat as one or more alerts. Each alert will augment the constructed model with a threshold probability value. When this value is exceeded, Hackystat will generate an email to the developer that the corresponding module has been predicted to be fault-prone, similar to the Complexity Threshold Alert email illustrated in Figure 3. The corresponding URL will show the values of the internal variables that gave rise to the alert. Evaluation of the model in the context of ongoing development will have both quantitative and qualitative components. The quantitative component will compare the model's predictions of fault-proneness to the actual ARs and IARs subsequently generated and test for significance using the PRESS (Predictive Sum of Squares) statistic [54]. The qualitative component will use buttons on each page displayed by alert URLs that allow the developer to give feedback to the system on the utility of that particular alert. In combination with developer interviews, qualitative feedback will provide insight into the perceived utility of the predictions to the developers, and ways in which in-process fault-proneness prediction can be improved.

5.4 Deliverables

The experiments will result in three classes of deliverables: technology, methodology, and measurement validation hypothesis testing. The principal technology deliverable is the enhanced version of the open source Hackystat system. The proposed research will expand Hackystat with a significant number of additional sensors for new tools and languages, as well as alerts built upon logistic regression modeling. As the research proceeds, we may investigate sensors for additional internal measures such as module-level documentation quality, which may predict certain types of faults for systems such as MDS which are intended to be used as components in other systems. The research will yield experience using a continuous, automated, in-process, and individualized methodology for measurement validation which has been used in five different experimental settings. This methodology will be made available for technology transfer within NASA as well as incorporated into software engineering educational settings using Hackystat. The final deliverable will be the results of hypothesis testing for each experiment, yielding insight into the kinds of internal measures useful for predicting fault-proneness (if any), the effect of developer experience (if any), and the development contexts in which the proposed approach to measurement validation is effective and usable for reducing dependability risks.

5.5 Timeline

This research project consists of five experiments, each of which has four phases. We plan to begin the first experiment in Fall, 2002, and anticipate each phase of the first experiment to require approximately six months to complete, for a total elapsed time of two years. Subsequent experiments can reuse and/or adapt sensors and models created during the first experiment, and this should reduce the interval time for experiments 2-5 to twelve to eighteen months. Experiments will be run concurrently after an initial startup period. Thus, we expect to complete the first two experiments in Fall, 2004, with the subsequent experiments completed every six months thereafter (Experiment 3, Spring 2005; Experiment 4, Fall 2005; Experiment 5, Spring 2006).

6 Anticipated contributions

The anticipated contributions of this research are organized according to the six review criteria for this program.

First, the research addresses the following fundamental research issues in dependable software-based computing and communication systems: (1) What product and process measures are useful for predicting the risk of fault-proneness and thus reducing dependability risks (if any)? (2) How can internal product and process measures be continuously collected and monitored at low cost and with minimal developer overhead? (3) How can fault-proneness prediction be integrated into in-process developer activities to support incremental product and process improvement?

Second, the research develops the following research products in the form of prototype tools or methodologies: (1) Sensors for collection of internal product and process measurements; (2) An XML and web-based storage, analysis, and retrieval service for product and process measures; (3) An email and web-based alert mechanism for communicating model results to developers in a just-in-time fashion; (4) A method for quantitative and qualitative evaluation;

Third, the research provides the following dependability attributes that are suitable for measuring the impact of the research products: (1) Quantitative risk models whose predictive accuracy has been evaluated in subsequent software release cycles; (2) Qualitative feedback from developers on the utility of the alerts for dependability risk reduction and the form of improvement chosen (inspection, redesign/refactoring, additional testing).

Fourth, the research provides a plan for the empirical evaluation/validation of the proposed research products: As illustrated by Figure 6, a set of five software measurement validation experiments will be undertaken during this research project and evaluated using the methodology described in Section 5.3.

Fifth, the research will have the following broader impacts: (1) Development of a sophisticated, freely available, open source software system for software measurement validation for use and adaptation by the software engineering community; (2) As the University of Hawaii is a university with 75% minority students in an EPSCOR state, this project will provide novel research opportunities to underrepresented groups. (3) Both empirical data and infrastructure technology will be made freely available via the Internet for broad dissemination.

Sixth, the intellectual merit of this proposal includes its novel combination of technology and methodology for assessing and improving the dependability of software. The proposed research builds upon two prior NSF-funded projects and upon research performed by the principle investigator over the past ten years. Access to the Mission Data System software and development organization at Jet Propulsion Laboratory and Testbed adaptation efforts will enable the proposed research to be evaluated in a high quality and high profile setting.

References Cited

[1] MDS: The strategic advantage for future flight projects. Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, January 2001.

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

[3] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability. Technical Report UCLA CSD Report 010028, University of California, Los Angeles, 2001.

[4] Victor Basili, Lionel Briand, and Walcelio Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10), 1996.

[5] Saida Benlarbi, Khaled El Emam, and Nishith Goel. Issues in validating object-oriented metrics for early risk prediction. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, November 1999.

[6] Aaron Binkley and Stephen Schach. Metrics for predicting run-time failures and maintenance effort: Four case studies. *CrossTalk*, August 1998.

[7] Aaron Binkley and Stephen Schach. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *Proceedings of the 20th Annual International Conference on Software Engineering*, 1998.

[8] S. Biyani and P. Santhanam. Exploring defect data from development and customer usage of software modules over multiple releases. In *Proceedings of the International Symposium on Software Reliability Engineering*, 1998.

[9] Barry Boehm. *Software Risk Management*. IEEE Computer Society Press, 1989.

[10] L. Briand, J. Wuest, J. Daly, and V. Porter. Exploring the relationships between design measures and software quality in object oriented systems. *Journal of Systems and Software*, 51, 2000.

[11] Lionel Briand, Jurgen Wust, and Hakim Lounis. Replicated case studies for investigating quality factors in object-oriented designs. *Journal of Empirical Software Engineering*, 2001.

[12] Teresa Cai, M. Lyu, K.-F. Wong, and M. Wong. ComPARE: A generic quality assessment environment for component-based software systems. In *Proceedings of the 2001 International Symposium on Information Systems and Engineering*, 2001.

[13] M. Cartwright. An empirical view of inheritance. *Information and Software Technology*, 40, 1998.

[14] Shyam Chidamber and Chris Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6), June 1994.

[15] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. Evaluating inheritance depth on the maintainability of object oriented software. *Journal of Empirical Software Engineering*, 1(2), 1996.

[16] Anne M. Disney and Philip M. Johnson. Investigating data quality problems in the PSP. In *Sixth International Symposium on the Foundations of Software Engineering (SIGSOFT'98)*, Orlando, FL., November 1998.

[17] Khaled El Emam. A methodology for validating software product metrics. Technical Report NRC/ERB-1076, National Research Council of Canada, June 2000.

[18] Khaled El Emam. Object-oriented metrics: A review of theory and practice. Technical Report NRC/ERB-1085, National Research Council of Canada, June 2000.

[19] Khaled El Emam, S. Benlarbi, N. Goel, and S. Rai. The confounding effect of class size on the validity of object oriented metrics. *IEEE Transactions on Software Engineering*, 27(7), 2001.

[20] Khaled El Emam, Walcelio Melo, and Javam Machado. The prediction of faulty classes using object oriented design metrics. *Journal of Systems and Software*, 56(1), 2001.

[21] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[22] Norman Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8), August 2000.

[23] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison-Wesley, 1993.

[24] R. Harrison, S. Counsell, and R. Nithi. Coupling metrics for object oriented design. In *Proceedings of the Fifth International Symposium on Software Metrics*, 1998.

[25] Allfusion harvest change manager. Computer Associates, Inc. (http://www.ca.com/).

[26] D. Hosmer and S. Lemeshow. *Applied Logistic Regression*. John Wiley and Sons, 1989.

[27] John Hudepohl, Stephen Aud, Tahgi Khoshgoftaar, Edward Allen, and Jean Mayrand. Emerald: Software metrics and models on the desktop. *IEEE Software*, 13(5), September 1996.

[28] Philip M. Johnson. Hackystat system. http://csdl.ics.hawaii.edu/Research/Hackystat/.

[29] Philip M. Johnson. Experiences with EGRET: An exploratory group work environment. *Collaborative Computing*, 1(1), January 1994.

[30] Philip M. Johnson. An instrumented approach to improving software quality through formal technical review. In *Proceedings of the 16th International Conference on Software Engineering*, May 1994.

[31] Philip M. Johnson. Supporting technology transfer of formal technical review through a computer supported collaborative review system. In *Proceedings of the Fourth International Conference on Software Quality*, Reston, VA., October 1994.

[32] Philip M. Johnson. Design for instrumentation: High quality measurement of formal technical review. *Software Quality Journal*, 5(3):33–51, March 1996.

[33] Philip M. Johnson. Reengineering inspection: The future of formal technical review. *Communications of the ACM*, 41(2):49–52, February 1998.

[34] Philip M. Johnson. Leap: A "personal information environment" for software engineers. In *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, CA., May 1999.

[35] Philip M. Johnson and Anne M. Disney. The personal software process: A cautionary case study. *IEEE Software*, 15(6), November 1998.

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

[37] Philip M. Johnson, Carleton A. Moore, Joseph A. Dane, and Robert S. Brewer. Empirically guided software effort guesstimation. *IEEE Software*, 17(6), December 2000.

[38] Philip M. Johnson and Danu Tjahjono. Assessing software review meetings: A controlled experimental study using CSRS. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 118–127, Boston, MA., May 1997.

[39] Philip M. Johnson and Danu Tjahjono. Does every inspection really need a meeting? *Journal of Empirical Software Engineering*, 4(1):9–35, January 1998.

[40] Philip M. Johnson, Danu Tjahjono, Dadong Wan, and Robert S. Brewer. Experiences with CSRS: An instrumented software review environment. In *Proceedings of the Pacific Northwest Software Quality Conference*, October 1993.

[41] Greg Kaszycki. Using process metrics to enhance software fault prediction models. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, November 1999.

[42] Taghi Khoshgoftaar and Edward Allen. Logistic regression modeling of software quality. *International Journal of Reliability, Quality and Safety Engineering*, 6(4), 1999.

[43] M. Lyu, J. Yu, E. Keramides, and S. Dalal. ARMOR: Analyzer for reducing module operational risk. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, 1995.

[44] Jean Mayrand and Francois Coallier. System acquisition based on software product assessment. In *Proceedings of the 18th International Conference on Software Engineering*, 1996.

[45] Kenny Meyer and Daniel Dvorak. Mission data system HDCP testbed: An end-to-end platform for improving the dependability of state-based mission software built from component frameworks. Technical report, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, 2002.

[46] Kenny Meyer and Carl Puckett. Introduction to the MDS process. Technical report, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, 2001.

[47] K-H Moller and D. Paulish. An empirical investigation of software fault distribution. In *Proceedings of the First International Software Metrics Symposium*, 1993.

[48] Carleton A. Moore. Project LEAP: Addressing measurement dysfunction in review. In *Proceedings of the Eighth International Conference on Human-Computer Interaction*, Munich, Germany, August 1999.

[49] G.J. Myers. *The Art of Software Testing*. John Wiley and Sons, New York, 1979.

[50] International Standards Organization. ISO/IEC 14598-2 information technology – product evaluation, part 1, general overview.

[51] Ganesh Pai and Joanne Dugan. Enhancing software reliablity estimation using bayesian belief networks and fault trees. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, November 2001.

[52] David L. Parnas and David M. Weiss. Active design reviews: Principles and practices. *Proceedings of Eighth International Conference on Software Engineering*, pages 132–136, August 1985.

[53] Adam A. Porter and Philip M. Johnson. Assessing software review meetings: Results of a comparative analysis of two experimental studies. *IEEE Transactions on Software Engineering*, 23(3):129–145, March 1997.

[54] R. Szabo and T. Khoshgoftaar. Modeling software quality in an object-oriented software system. In *Proceedings of the Annual Oregon Workshop on Software Metrics*, 1995.

[55] M.-H. Tang, M.-H. Kao, and M.-H. Chen. An empirical study on object oriented metrics. In *Proceedings of the Sixth International Symposium on Software Metrics*, 1999.