

## White Paper

**Improving the dependability and predictability of JPL/MDS software  
through low-overhead, automated collection and analysis of software process and product metrics**

Philip Johnson  
Collaborative Software Development Laboratory  
Department of Information and Computer Sciences  
University of Hawaii

Last Update: 05/20/2002 11:57 AM

**1. Introduction**

The goals of this white paper are to:

1. Provide background information useful for determining approaches to collaboration between JPL/MDS, USC/CSE, and UH/CSDL.
2. Outline a research project direction compatible with the joint NSF/NASA [Highly Dependable Computing and Communication Systems Research \(HDCCSR\) program solicitation](#);
3. Provide a strawman document to facilitate further discussion between the three organizations.

Section 2 provides a high level overview of the goals of the HDCCSR program, selected because HDCCSR appears to be the funding opportunity with the most immediate deadline (July 4, 2002). The HDCCSR program goals appear to be quite congruent with the proposed approach to collaboration, but to exploit this opportunity, we will need to begin work on the proposal very soon. Section 3 presents a brief overview of several aspects of MDS software development that appear relevant to this collaboration. Section 4 presents a brief overview of several aspects of the Hackstat research project that appear relevant to this collaboration. Section 5 presents a possible direction for joint research that appears compatible with the HDCCSR program, MDS software development, UH/CSDL, and USC/CSE. Finally, Section 6 lists some issues for future discussion.

**2. Overview of the NSF/NASA HDCCSR Program**

Here are some relevant excerpts from the program solicitation:

- NSF and NASA will cooperate to fund projects that will promote the ability to design, test, implement, evolve, and certify highly dependable software-based systems. A significant feature of this solicitation is the use of a new NASA test-bed facility that will allow researchers to experimentally evaluate their research products on significant real hardware/software artifacts.
- The overall goal of this solicitation is to develop a scientific basis for measurable and predictable dependability in software-based computing and communication systems, and a scientific basis - comparable to those in physics-based engineering disciplines - for technologies or methodologies to improve dependability in these systems.
- A successful scientific demonstration would show that objective, dependability attributes that are measurable before deployment can be used to predict measurable, dependability attributes of deployed systems; similarly, a successful technology or methodology demonstration would show that a technology or methodology has a measurable impact on the dependability of deployed systems, and this impact can be predicted before deployment.
- Proposals funded under this solicitation must meet four requirements. They must:
  1. address fundamental research issues in dependable software-based computing and communication systems,
  2. develop research products in the form of prototype tools or methodologies,
  3. provide dependability attributes that are suitable for measuring the impact of the research products, and
  4. include a plan for the empirical evaluation/validation of the proposed research products.

- Proposals of up to \$160,000 per year for up to 4 years are sought from eligible institutions, as described in the NSF Grant Proposal Guide. Proposal evaluation will be done by a standard NSF peer review process followed by a NASA evaluation for appropriateness and relevance to NASA objectives.

### 3. A brief overview of MDS Software Development

MDS uses a high quality incremental development process with excellent tool support for configuration management, system build, and automated testing. The current level of automation creates many opportunities for the sensor-based approach utilized by Hackstat. To gain a sense for collaboration, it is helpful to understand the representation of work packages, the MDS CCC-Harvest state model for work package workflow management, and some ideas regarding dependability in MDS software development and how it might be improved.

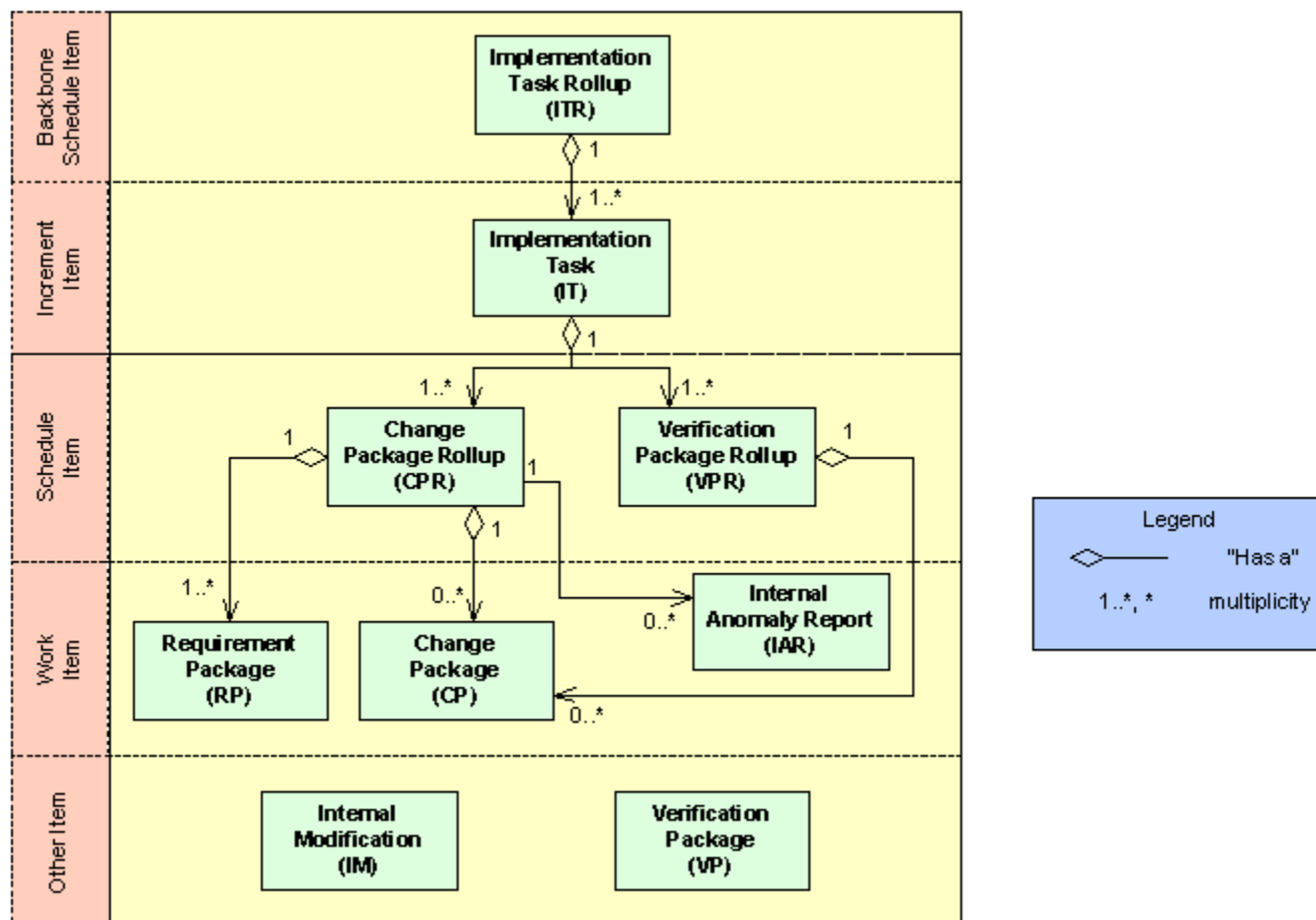
The illustrations in this section are taken from the PowerPoint presentations "Introduction to MDS Process", by Kenny Meyer and Carl Puckett and "MDS Build Process Metrics", by Carl Puckett.

#### 3.1 The work package model

At MDS, each mission scenario or incremental development of an Architectural Element corresponds to an "Implementation Task Rollup (ITR)". Each ITR consists of a set of "Implementation Tasks (ITs)". Each IT consists of a set of "Change Package Rollups (CPRs)". A CPR is the top-level unit of schedulable work. A CPR consists of a set of Change Packages (CPs) and their corresponding Verification Packages (VPs).

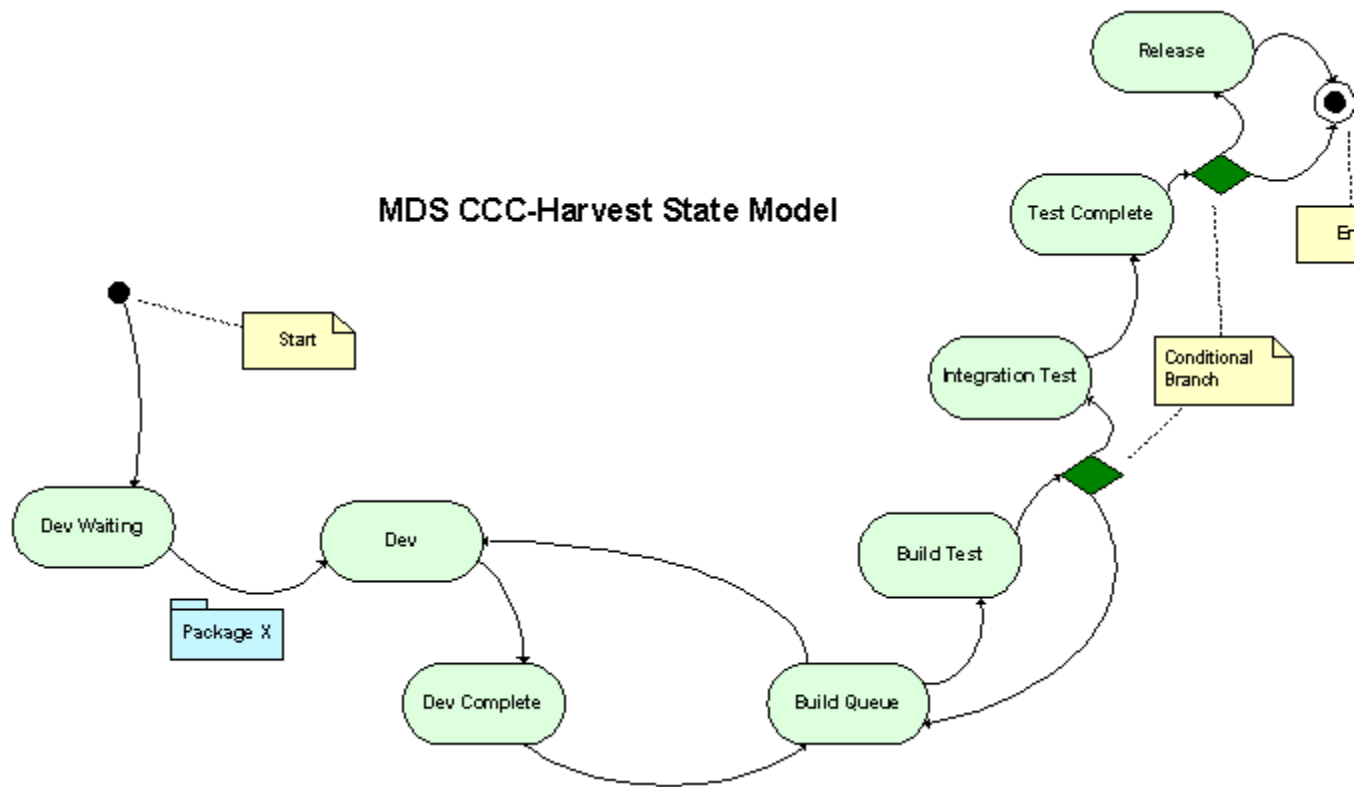
The preceding work packages represent work that is generated "top-down" through planning and requirements specification processes. To represent work that is generated "bottom-up" through defects and other unexpected problems, MDS provides "Internal Anomaly Reports (IARs)" and "Internal Modifications (IMs)".

Here is an illustration of the components of the work package model and their relationships:



### 3.2 The MDS CCC/Harvest State Model

Implementation Task Rollups and their associated Implementation Tasks, as well as Change/Verification Package Rollups are tracked manually using MS Project. Automated support for managing and tracking the progress of development begins at the level of Change Packages, where physical files and resources are bound to the Change Package representation. MDS uses [CCC/Harvest](#) to support development. This tool provides integrated support for configuration management, system building, and testing (think CVS + Make), along with a workflow management tool that supports tracking of change packages as they proceed through the modification, build, and test process. The following illustration shows the MDS workflow:



As you can see, Change Packages are defined in the "Dev Waiting" state. Actual development on a Change Package occurs in "Dev". Once completed, the Change Package moves into the "Build Queue", where an attempt is made to build the system using the Change Package and the most recent baseline version of the system. If that is successful, then the newly built system moves to the "Build Test" state where it is tested against the tests in the Verification Package corresponding to this Change Package. If that is successful, then the newly built system is tested against the entire test suite in the "Integration Test" state. If that is successful, then the newly built system becomes the new baseline into which future Change Packages are built. The following summarizes each of these states:

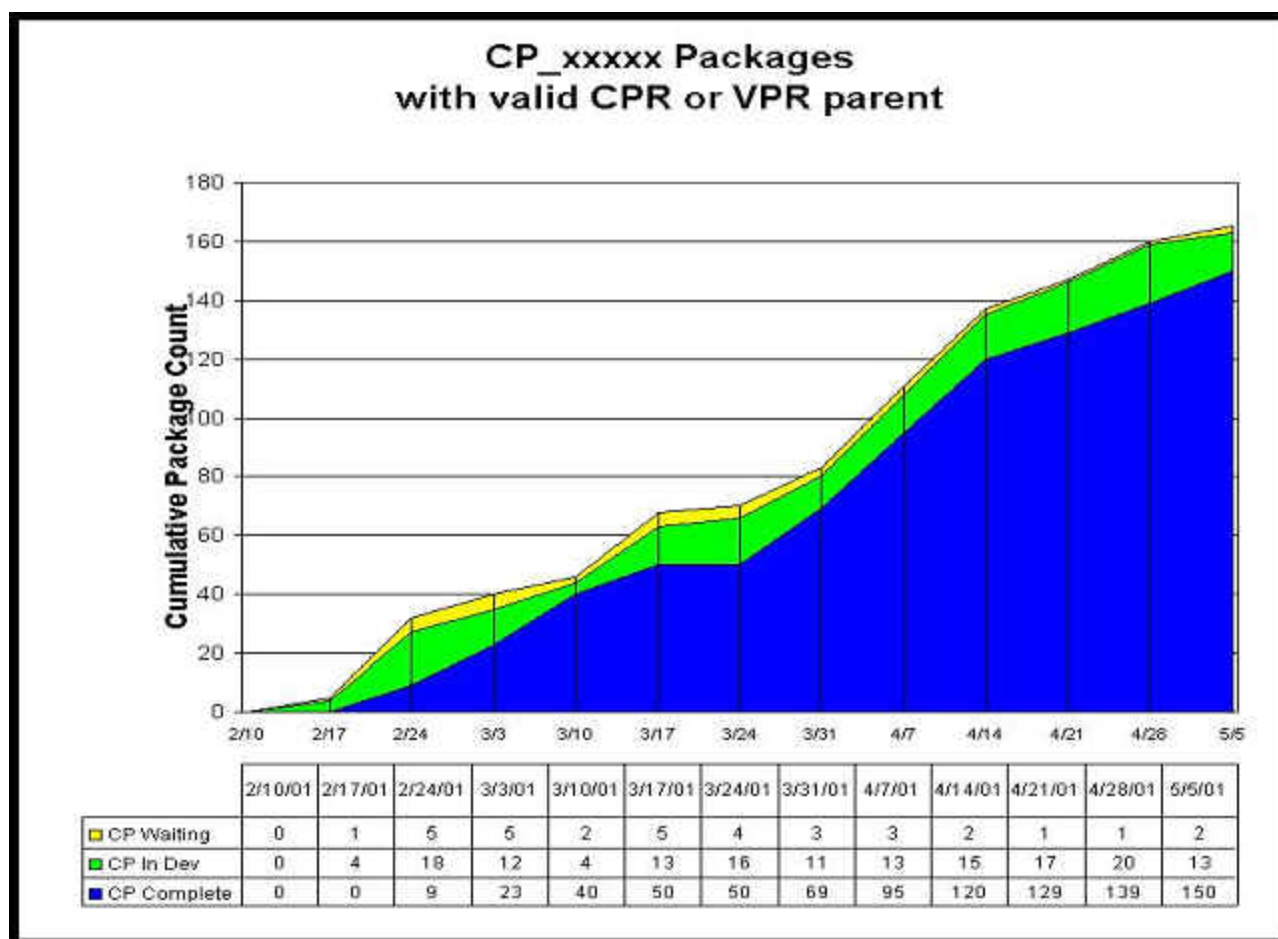
State	Description
Dev Waiting	Packages in the Dev Waiting state serve as placeholders. No artifacts can be attached to a package in Dev Waiting.
Dev	Packages in Dev have artifacts that are under development. A child package can only be promoted to Dev if its parent is also in Dev. <i>NOTE: Since Harvest uses a delta-based repository, artifacts that remain in Dev are subject to entanglement and should be promoted to Dev Complete as quickly as possible.</i>
Dev Complete	Packages in Dev Complete have artifacts that are complete. A parent package can only be promoted to Dev Complete if all its children are in Dev Complete or higher states.
Build Queue	Packages in the Build Queue have been promoted by the Build Manager of the CML team in preparation for compiling, linking and testing. The order of the packages in the Build Queue reflects physical dependencies among the packages.
Build Test	Packages in the Build Test are currently being compiled, linked and tested.
Integration Test	Packages in Integration Test have artifacts that are part of a baseline.
Test Complete	Packages in Test complete include artifacts that have been subject to verification.
Release	Packages in Release include artifacts that are part of a released product (at the conclusion of the outer loop).

Of course, things don't always work that smoothly. Some of the potential breakdowns in the process include:

- **Entanglement:** the need by two Change Packages in the Dev state to modify the same file at the same time.
- **Build failure:** the inability to compile or link the Change Package files against the most recent baseline build.
- **Build Test failure:** the inability of the newly built system to pass the Verification tests associated with the current Change Package.
- **Integration Test failure:** the inability of the newly built system to pass the full suite of test cases.

Resolving entanglement requires either sequentializing development of the two Change Packages (which may slow down development) or else incurring additional coordination costs on development.

Resolving any of the three types of failures requires rework, which is managed through the generation of either an Internal Modification or Internal Anomaly Report. The impact of a failed build on overall development can be mild or severe. Carl told me that the development group refers to particularly problematic builds as "Pain Packages", and that they can bring the normal Change Package throughput to a complete halt for extended periods of time. Carl referred me to the following chart, which reveals one such Pain Package during the week of 3/17 to 3/24. Note that the number of newly completed Change Packages during that week dropped to zero.



What makes a package painful is not simply that it takes a long time to push from Build Queue state to the Release state. In addition, (a) pain packages are not recognized before the pain is actually being inflicted, and (b) the length of time that the pain is inflicted is unpredictable. In this case, productivity (in the sense of Change Package throughput) came to a complete and unanticipated halt for a week.

### 3.3 Improving dependability in MDS

It is important to note that without any changes to their current procedures, MDS appears already able to achieve arbitrarily high levels of dependability in their software. The problem is that developing such software might take MDS an arbitrarily long time to finish, and it might cost an arbitrarily large amount of money. A possible goal of this research collaboration is to help MDS develop highly dependable software more quickly, more cheaply, and more predictably.

Given the above development procedures, success in some combination of the following areas would be likely to yield improved dependability:

- Increased Change Package throughput.
- Improved predictability of Change Package throughput.
- Decrease in the rate of failure during Build Queue, Build Test, and Integration Test.
- Decrease in the average time required for "recovery" from a failure during Build Queue, Build Test, and Integration Test.
- Increased Implementation Task Rollup throughput
- Improved predictability of Implementation Task Rollups

These can be viewed as a first pass at measurable goals for a dependability improvement initiative. Note that it is probably possible to improve in any one area without actually improving dependability. For example, one can increase Change Package throughput by simply reducing the size and scope of Change Packages, but simply reducing the size and scope of Change Packages probably will not by itself improve dependability.

If we believe that these goals, rationally pursued, would lead to improved dependability, then we could pose the following research question: how can MDS make progress toward these goals without incurring additional cost on development and/or increasing the time required for development? Indeed, might there even be a way to improve dependability while simultaneously *decreasing* the cost and time

required to develop MDS software?

I propose that these goals and the questions of time and cost surrounding their achievement might form the basis of a fundable proposal to the HDCCSR program. Furthermore, I would like to propose the following research strategy for achieving these goals:

1. Develop instrumentation for the MDS software development procedures that automatically captures these metrics. Before we can begin any improvements, we need baseline measures and the ability to determine the impact of change on the measures.
2. Develop additional instrumentation for the MDS software development procedures that supports the process of root cause analysis. In other words, we need to go further than simply determining the average time required for to recover from a Build Test failure, which is what (1) provides. We need additional information about the events leading up to each of the Build Test failures that helps developers determine why some Build Test failures require a small amount of recovery time while others require a large amount of recovery time, and what developers need to do differently in order to avoid large recovery times in future.
3. Once the root cause analysis of a problem has taken place, automate (when possible and appropriate) the detection of the kinds of events and event sequences that tend to co-occur with the later occurrence of a problem. In other words, develop a kind of "early warning system" that makes developers aware of the possibility of problems early on when relatively cheap preventative measures might be possible and when schedule adjustments can be made more easily.
4. Use the data collected through the instrumentation as additional input to a Cocomo-style cost modelling tool to provide more accurate predictions of the cost in time and resources to develop dependable software in this environment.

Before describing additional instrumentation, let's look at the instrumentation that is already available for automated metrics collection and analysis. Current instrumentation basically consists of the log files produced by CCC/Harvest that indicate when each Change Package enters and exits each state, along with a high-level indication of why that state transition occurred. This log provides useful but relatively "coarse-grained" measures of development, as illustrated below:

<u>CURRENT STATE</u>	<u>PACKAGE ID</u>	<u>DATE ENTERED STATE</u>	<u>STATE</u>	<u>ACTION</u>
Test Complete	CP-00001	2000/10/03:23:59	Dev	Created
Test Complete	CP-00001	2000/10/11:00:22	Dev Complete	Promoted from Dev
Test Complete	CP-00001	2000/10/11:00:40	Dev	Demoted from Dev Complete
Test Complete	CP-00001	2000/10/11:00:43	Dev Complete	Promoted from Dev
Test Complete	CP-00001	2000/10/17:15:39	Subsystem Test	Promoted from Dev Complete
Test Complete	CP-00001	2000/10/30:23:20	Integr Test	Promoted from CM Build
Test Complete	CP-00001	2000/11/16:20:25	CM Build	Demoted from Integr Test
Test Complete	CP-00001	2000/11/16:20:27	Integr Test	Promoted from CM Build
Test Complete	CP-00001	2001/02/14:17:03	Test Complete	Promoted from Integr Test
Test Complete	CP-00001	2001/02/14:21:46	Test Complete	Modified
Test Complete	CP-00002	2000/10/04:00:12	Dev	Created
Test Complete	CP-00002	2000/10/11:00:24	Dev Complete	Promoted from Dev
Test Complete	CP-00002	2000/10/17:15:39	Subsystem Test	Promoted from Dev Complete
Test Complete	CP-00002	2000/10/30:23:20	Integr Test	Promoted from CM Build
Test Complete	CP-00002	2001/02/14:17:03	Test Complete	Promoted from Integr Test
Test Complete	CP-00003	2000/10/04:18:09	Dev	Created
Test Complete	CP-00003	2000/10/11:00:27	Dev Complete	Promoted from Dev
Test Complete	CP-00003	2000/10/17:15:39	Subsystem Test	Promoted from Dev Complete
Test Complete	CP-00003	2000/10/30:23:20	Integr Test	Promoted from CM Build
Test Complete	CP-00003	2001/02/14:17:03	Test Complete	Promoted from Integr Test
Test Complete	CP-00006	2000/10/05:18:36	Dev	Created
Test Complete	CP-00006	2000/10/11:00:28	Dev Complete	Promoted from Dev
Test Complete	CP-00006	2000/10/17:15:39	Subsystem Test	Promoted from Dev Complete
Test Complete	CP-00006	2000/10/30:23:20	Integr Test	Promoted from CM Build
Test Complete	CP-00006	2001/02/14:17:03	Test Complete	Promoted from Integr Test

To pursue the research strategy that would result in support for both root cause analysis and eventual automated detection of abnormal conditions, I believe that these useful but "coarse grained" measures provided by CCC/Harvest should be augmented with "fine grained" measures. Furthermore, I believe that these fine grained measures must be automatically collected and analyzed, otherwise they won't be. Finally, I would like to propose Hackystat as useful infrastructure for the development of these fine grained measures.

#### 4. A brief overview of the Hackstat Project

Hackstat is an approach to low-cost collection and analysis of fine-grained software project and product data. The design of Hackstat arises from our prior research, in which we observed (among other things) that:

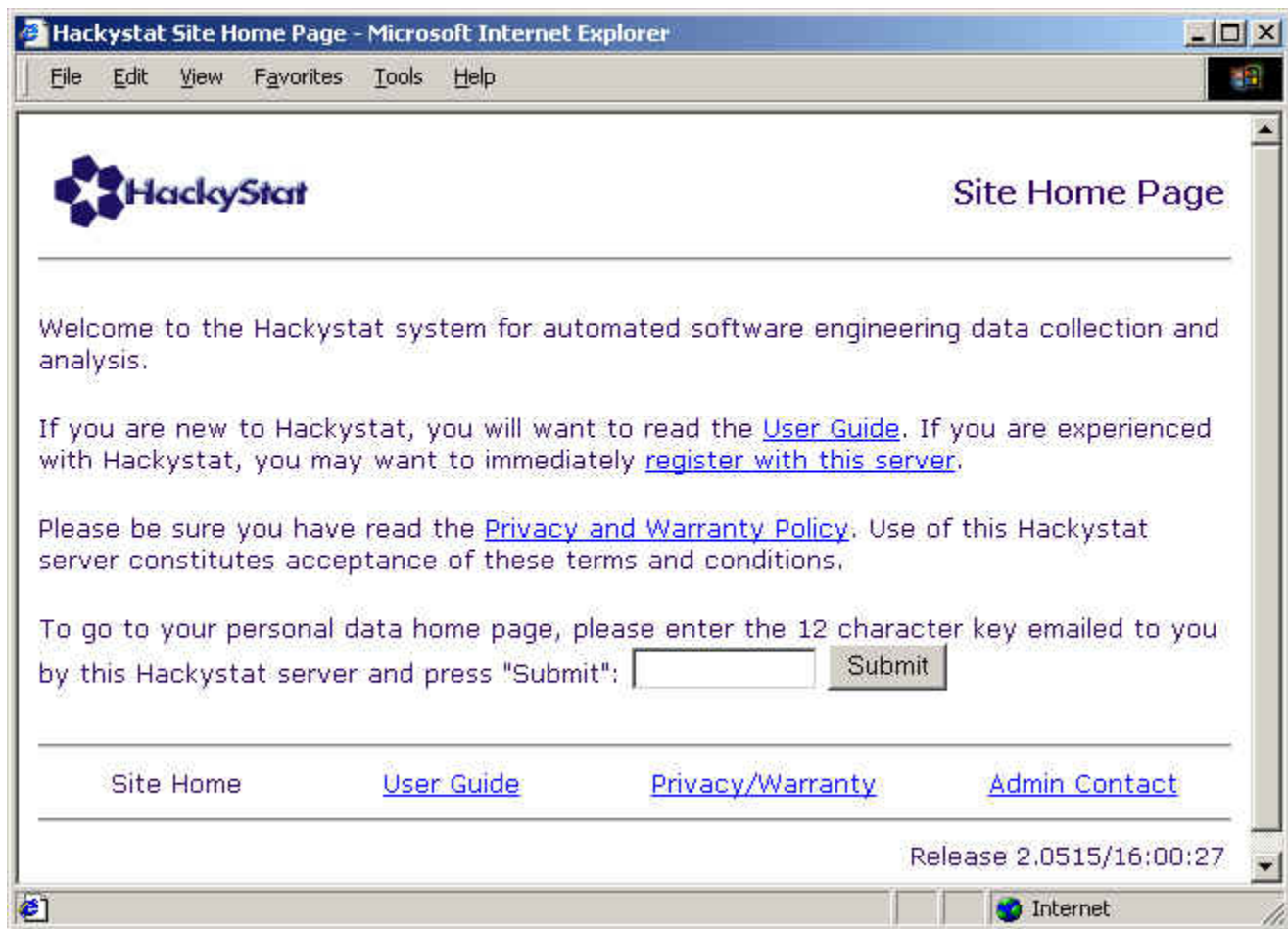
- **Fine-grained project and process data can provide significant insight to developers.** For example, with data concerning the size of the work products a developer creates, the time it takes the developer to create them, and the defects that result, a developer can improve the quality of their work products (by better understanding the sources of defects) and the predictability of development (by better understanding how long development takes).
- **It is difficult to lower the overhead of fine-grained project and process data collection and analysis to an acceptable level, regardless of the ensuing benefits to developers.** For example, in 1996, I taught a version of the Personal Software Process in which all data collection and analysis was manual (using spreadsheets). Despite substantial and demonstrable achievements in size estimation, time estimation, and defect reduction, no students continued to use the method after the semester ended. In 1999, I taught a new version of the course, in which we employed a kind of "Process Dashboard" (called [Leap](#)) which substantially reduced the overhead of time, size, and defect data collection and analyses while preserving the benefits. However, developers still needed to manually invoke the collection and analysis tools, and we found that even a seemingly minimal level of in-process overhead was still enough to undermine adoption of the tool by all but a few of the most "disciplined".

As a result of these experiences, in summer 2001 we began the design of a new approach to fine-grained data collection and analysis called Hackstat. The first fundamental design principle of Hackstat is that developers should incur no overhead for in-process data collection and analysis. To use Hackstat, developers install "sensors" into their development tools, such as Emacs, JBuilder, Visual Studio, CVS, JUnit, and so forth. These tools silently and automatically watch developer activity and send information off to a centralized web service. Analyses are periodically and automatically invoked on the developer data by the web service, and when "things of interest" are observed, the web service sends an email to the developer informing them of the presence of some form of anomaly or unexpected condition in their data, and providing them with a link into the web service where they can learn more about the data that caused the anomaly as a first step toward taking any required corrective action.

The following sections briefly overview the current status of the system.

##### 4.1 Registration and installation

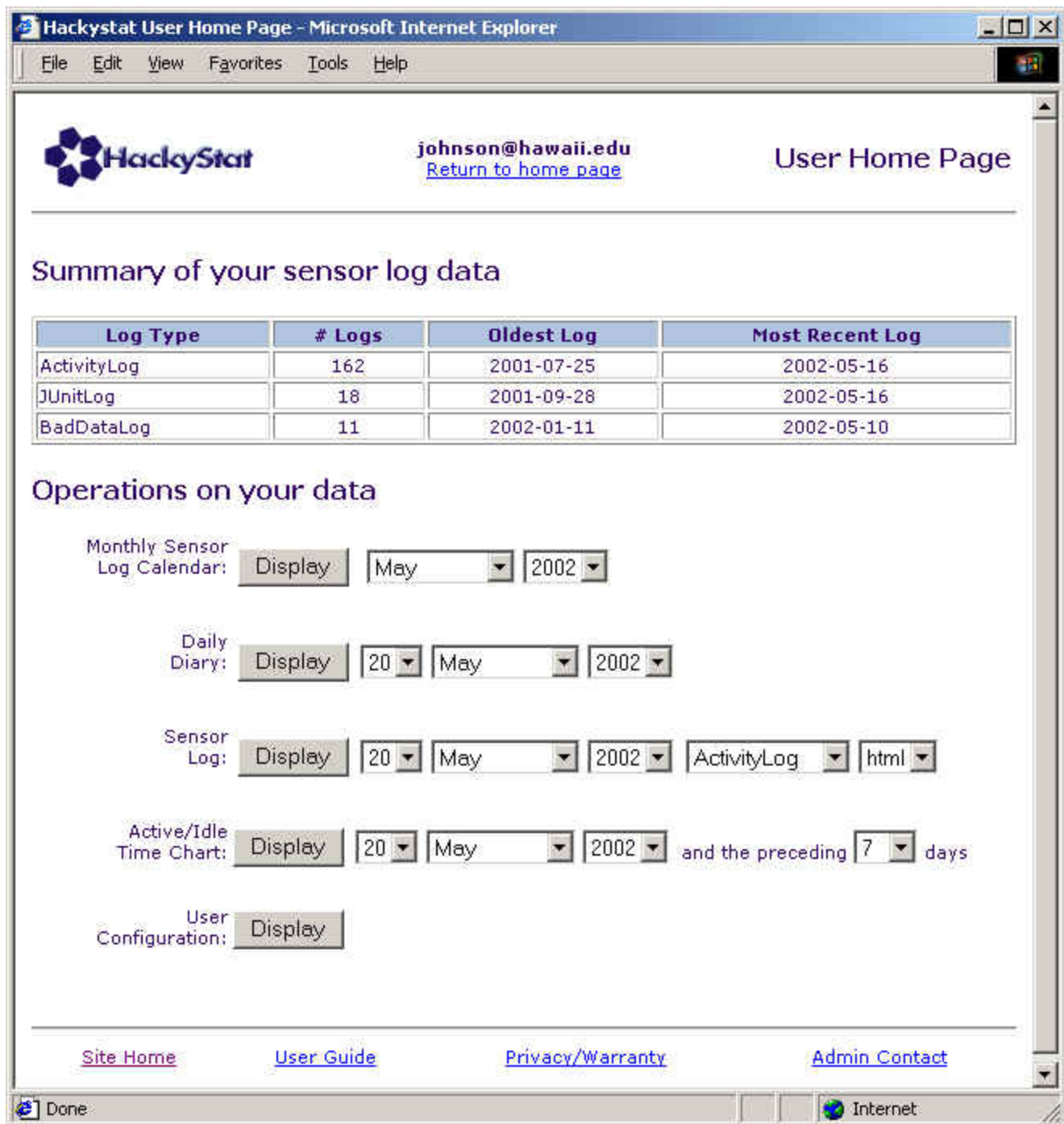
The first screen image below shows the server home page. A developer goes here to register with the service and provide an email address to which the server sends notifications when it observes events of interest. The developer also downloads any sensors they choose to install from the web server. So far, we have implemented sensors for Emacs, JBuilder and JUnit (a Java-based unit testing tool). We intend to implement additional sensors to track CVS activity and the size of software artifacts this summer, with additional sensors scheduled for the Fall and beyond.



Although we maintain a public web server that anyone can use, we expect that MDS will prefer to run their own Hackystat server behind their firewall.


Once registered, each user is sent a 12 character key via email that they can use to access their home page. One such home page is illustrated below:





**Hackystat User Home Page - Microsoft Internet Explorer**

File Edit View Favorites Tools Help

 [johnson@hawaii.edu](mailto:johnson@hawaii.edu) [Return to home page](#) **User Home Page**

---

### Summary of your sensor log data

Log Type	# Logs	Oldest Log	Most Recent Log
ActivityLog	162	2001-07-25	2002-05-16
JUnitLog	18	2001-09-28	2002-05-16
BadDataLog	11	2002-01-11	2002-05-10

### Operations on your data

Monthly Sensor Log Calendar:

Daily Diary:

Sensor Log:

Active/Idle Time Chart:     and the preceding  days

User Configuration:

---

[Site Home](#) [User Guide](#) [Privacy/Warranty](#) [Admin Contact](#)

Done Internet

#### 4.2 Sensor-based process and product data collection

Once the sensors have been downloaded and installed, the developers can return to real work. The only difference developers will notice in their tools is a message informing them when data is sent to a server. Data transmission typically takes two to three seconds, and is generally timed to occur when the tool is otherwise idle. Here is a screen image displaying this notification message in the Emacs minibuffer:

```

emacs@THERESA
File Edit Options Buffers Tools Insert Help

if ($?ANT_OPTS == 0) then
    setenv ANT_OPTS ""
endif

if ($?CATALINA_OPTS == 0) then
    setenv CATALINA_OPTS ""
endif

# Unset CLASSPATH in this shell to avoid configuration problems.
setenv CLASSPATH ""

# Set up OPTS variables.
setenv HACKYSTAT_OPTS "-Dhackystat_host=$HACKYSTAT_HOST -Dhackystat_admin_email=%$i
$TAT_ADMIN_EMAIL -Dhackystat_mail_server=$HACKYSTAT_MAIL_SERVER -Djbuilder_home=$JBI
$ _HOME -Dcatalina_home=$CATALINA_HOME"

setenv ANT_OPTS "$HACKYSTAT_OPTS $ANT_OPTS"
setenv CATALINA_OPTS "$HACKYSTAT_OPTS $ANT_OPTS"

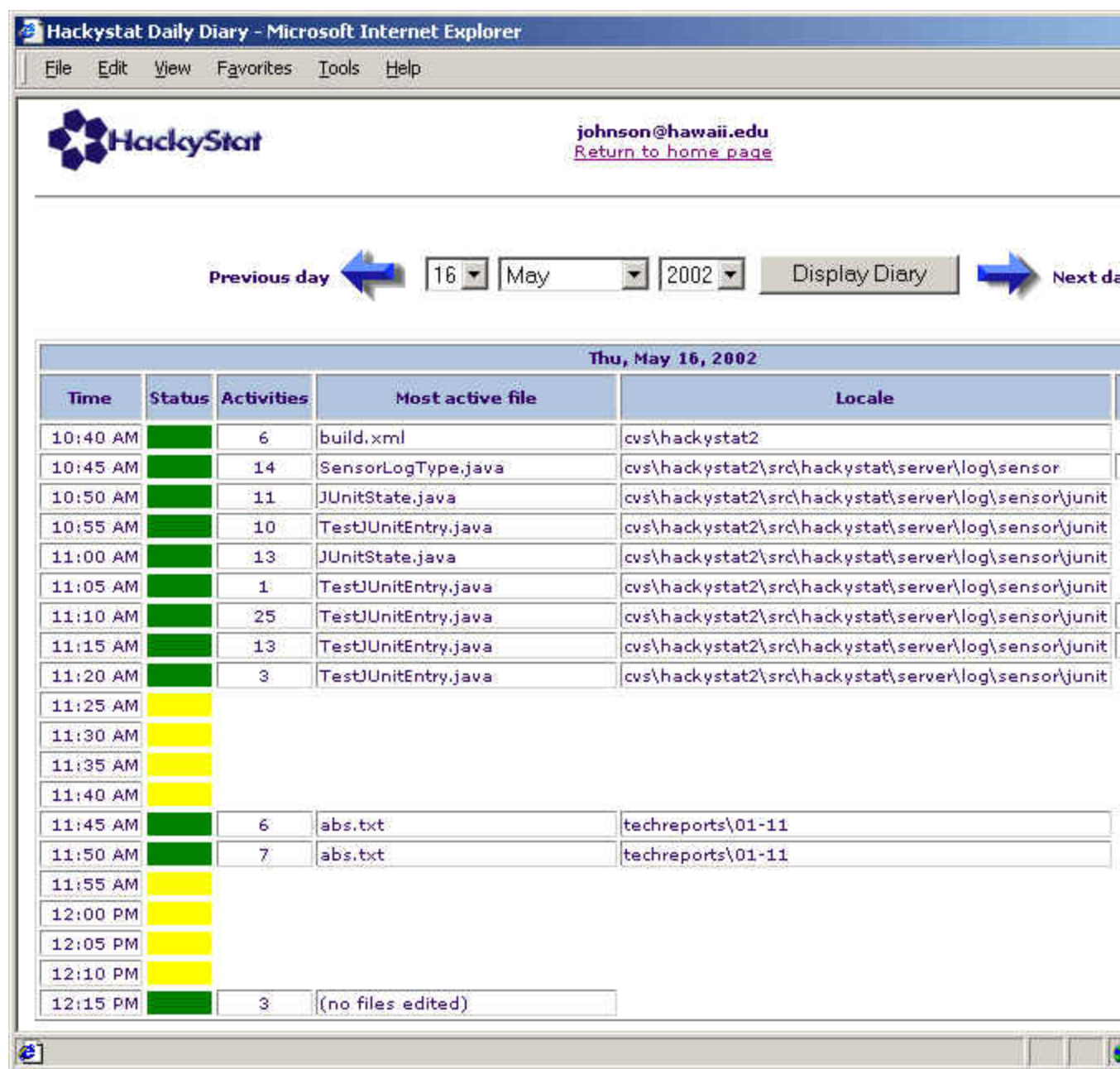
# Assume 'ant' will always be invoked in the top-level directory.
setenv ANT_HOME "./lib/Ant1.4.1"
--(Unix)-- setEnv.csh (Shell-script[csh] CVS:1.1)--L52--54%-----
(Hackystat) Sending data on 4 activities to http://katrina.ics.hawaii.edu/...done.

```

The events tracked by editor sensors such as those for Emacs and JBuilder include file opening and closing, invocation of the compiler, and so forth. The JUnit sensor sends data regarding the set of test cases invoked and their result (pass, fail, error). A CVS sensor (not yet implemented) could track the files that were checked out, committed, tagged, etc. by a developer. A CCC/Harvest sensor (not yet implemented) could make the data in the log file available for subsequent analysis. Over the course of a day's work by a single developer, data on dozens to hundreds of development events might be collected by different sensors.

### 4.3 Server-based sensor data analysis

One basic analysis task is to distill out from the sensor event stream an understandable and reasonably accurate high-level record of what the developer was doing during the day. Our current approach is to break up each day into five minute intervals, and then assign to each five minute interval a single file which appears to represent the primary focus of attention of the developer during that interval. (The actual algorithm is described elsewhere and will be the subject of validation studies this summer.) The result is what we call the "Most Active File" which can be displayed by viewing the "Daily Diary" page. Here is an excerpt of a Daily Diary page for a short period of time that shows the "Most Active File", the "Locale" in which that file exists (its directory path), as well as another column indicating the receipt of JUnit sensor data during that same period of time:



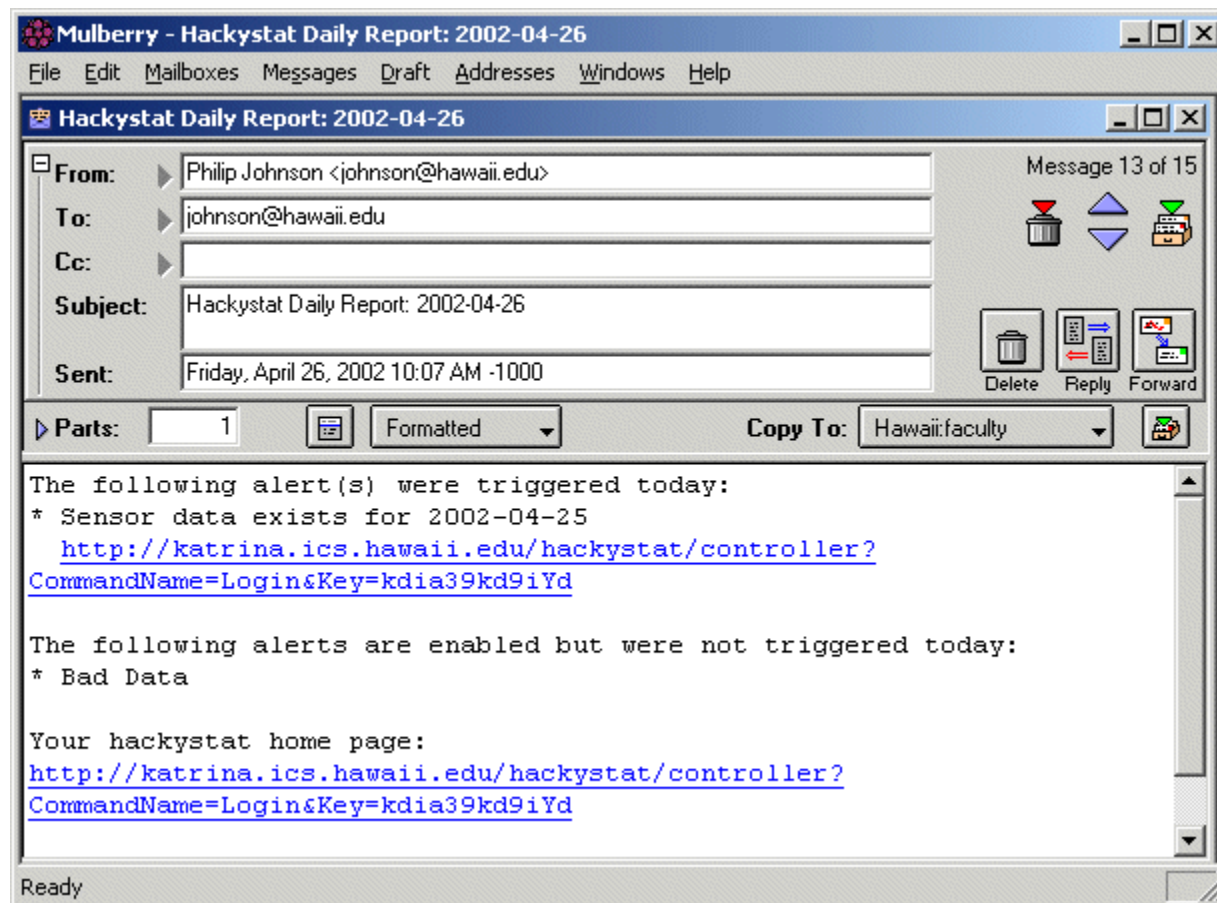
This Daily Diary page illustrates several important design decisions in the current implementation of Hackstat:

1. The approach of assigning a most active file to each five minute period provides an effort metric for work on single files and collections of files within the same subdirectory tree. This effort metric is different from "billable hours", in that it does not track time spent in meetings or other tasks outside of the use of sensor-enabled development tools.
2. By augmenting the sensors to collect structural properties (such as size and/or complexity) on the most active files, we can gather a body of data that developers can use to better understand and improve their development process. As an obvious example, Hackstat can provide the developer with information regarding their productivity in LOC/hour, along with additional information that enables them to better utilize this information for predictive purposes (such as their personal variability and how that might correlate with other properties of development, such as complexity, test results, etc.) More subtle analyses might relate metrics of test case coverage to resulting build success rates in an effort to optimize test development productivity.
3. We need to validate our approach to determining the Most Active File. We intend to perform an initial validation study this summer, in which we will videotape developers for hour long periods as they work and compare the video record to the Most Active File values computed by Hackstat.

In the longer term, this Daily Diary page illustrates the possibilities for inference via "sensor data fusion". The page shows that between 10:45am and 11:50am, the developer ran 50 JUnit tests, all of which passed. The developer began working on two files, JUnitState, and TestJUnitEntry, which lead to the running of 51 tests between 11:10am and 11:15am---one of which failed. After a few more minutes of work, all 51 tests now pass. By combining the two streams of data together, it might be possible to infer that the developer was writing developing new code and a single associated unit test for that code during this time. (In fact, this is exactly what I was doing that morning :-)

#### 4.4 Automated notification of "interesting" analyses

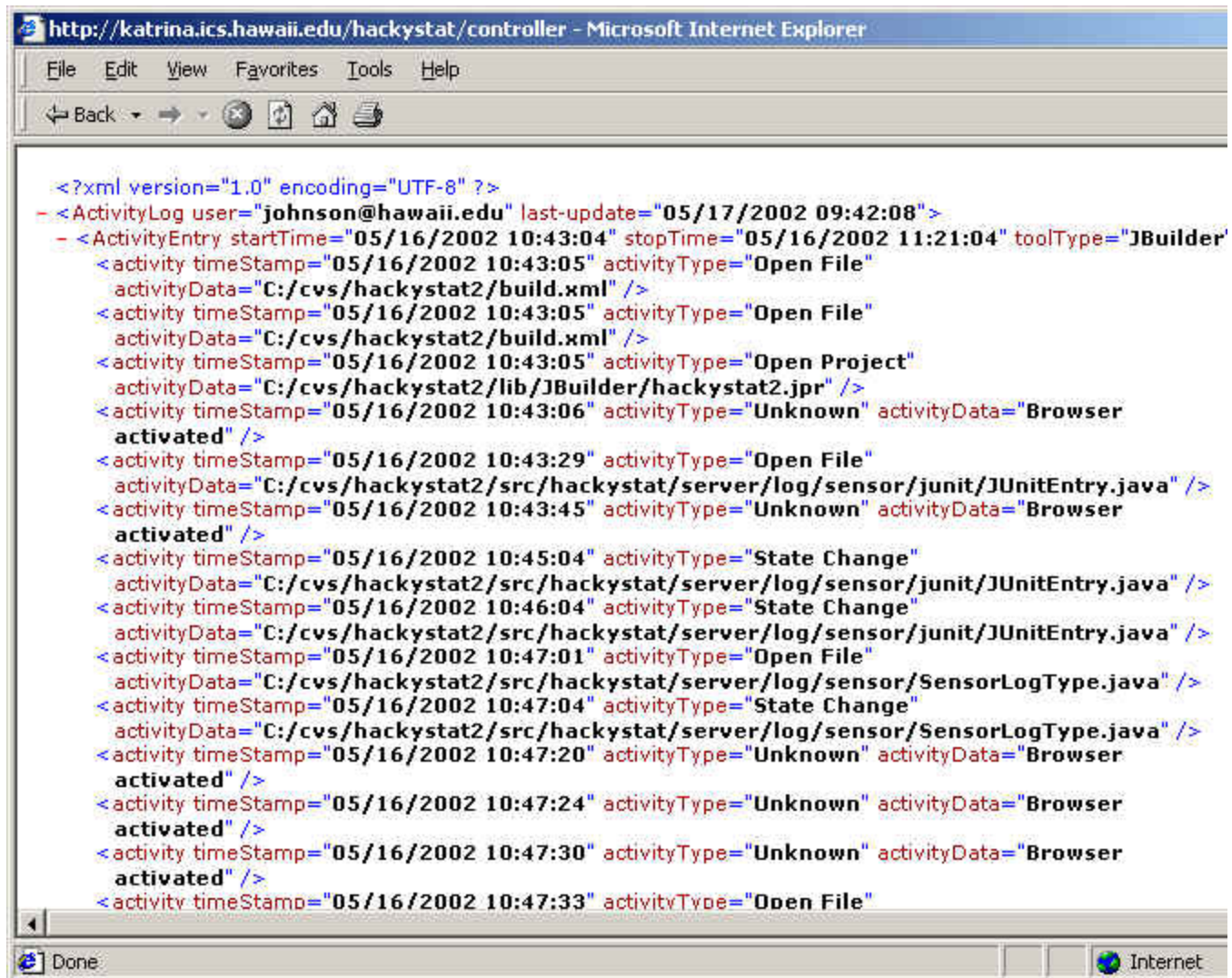
While analyses and abstractions such as the Daily Diary page will sometimes provide useful insight to developers, more often they will not. The second fundamental design principle of Hackystat is that developers must be automatically informed by the system when analyses result in information of potential interest. Our approach is to attempt to detect "anomalies" in the event stream (usually via analyses performed once per day) and then send the developer an email with an indication that an anomaly occurred and a link to a page that provides more detail. We are just starting the exploration of anomaly detection, but the infrastructure is already available and in use to provide developers with a daily email whenever they sent data to Hackystat on the previous day:



#### 4.5 Display and presentation of the raw sensor data stream

The Daily Diary page is the most usable interface to the data for a particular day, but it is also possible to see the "raw" event data. Here is a screen image of a portion of the data in a single sensor log:





```

<?xml version="1.0" encoding="UTF-8" ?>
- <ActivityLog user="johnson@hawaii.edu" last-update="05/17/2002 09:42:08">
- <ActivityEntry startTime="05/16/2002 10:43:04" stopTime="05/16/2002 11:21:04" toolType="JBuilder"
  <activity timeStamp="05/16/2002 10:43:05" activityType="Open File"
    activityData="C:/cvs/hackystat2/build.xml" />
  <activity timeStamp="05/16/2002 10:43:05" activityType="Open File"
    activityData="C:/cvs/hackystat2/build.xml" />
  <activity timeStamp="05/16/2002 10:43:05" activityType="Open Project"
    activityData="C:/cvs/hackystat2/lib/JBuilder/hackystat2.jpr" />
  <activity timeStamp="05/16/2002 10:43:06" activityType="Unknown" activityData="Browser
    activated" />
  <activity timeStamp="05/16/2002 10:43:29" activityType="Open File"
    activityData="C:/cvs/hackystat2/src/hackystat/server/log/sensor/junit/JUnitEntry.java" />
  <activity timeStamp="05/16/2002 10:43:45" activityType="Unknown" activityData="Browser
    activated" />
  <activity timeStamp="05/16/2002 10:45:04" activityType="State Change"
    activityData="C:/cvs/hackystat2/src/hackystat/server/log/sensor/junit/JUnitEntry.java" />
  <activity timeStamp="05/16/2002 10:46:04" activityType="State Change"
    activityData="C:/cvs/hackystat2/src/hackystat/server/log/sensor/junit/JUnitEntry.java" />
  <activity timeStamp="05/16/2002 10:47:01" activityType="Open File"
    activityData="C:/cvs/hackystat2/src/hackystat/server/log/sensor/SensorLogType.java" />
  <activity timeStamp="05/16/2002 10:47:04" activityType="State Change"
    activityData="C:/cvs/hackystat2/src/hackystat/server/log/sensor/SensorLogType.java" />
  <activity timeStamp="05/16/2002 10:47:20" activityType="Unknown" activityData="Browser
    activated" />
  <activity timeStamp="05/16/2002 10:47:24" activityType="Unknown" activityData="Browser
    activated" />
  <activity timeStamp="05/16/2002 10:47:30" activityType="Unknown" activityData="Browser
    activated" />
  <activity timeStamp="05/16/2002 10:47:33" activityType="Open File"

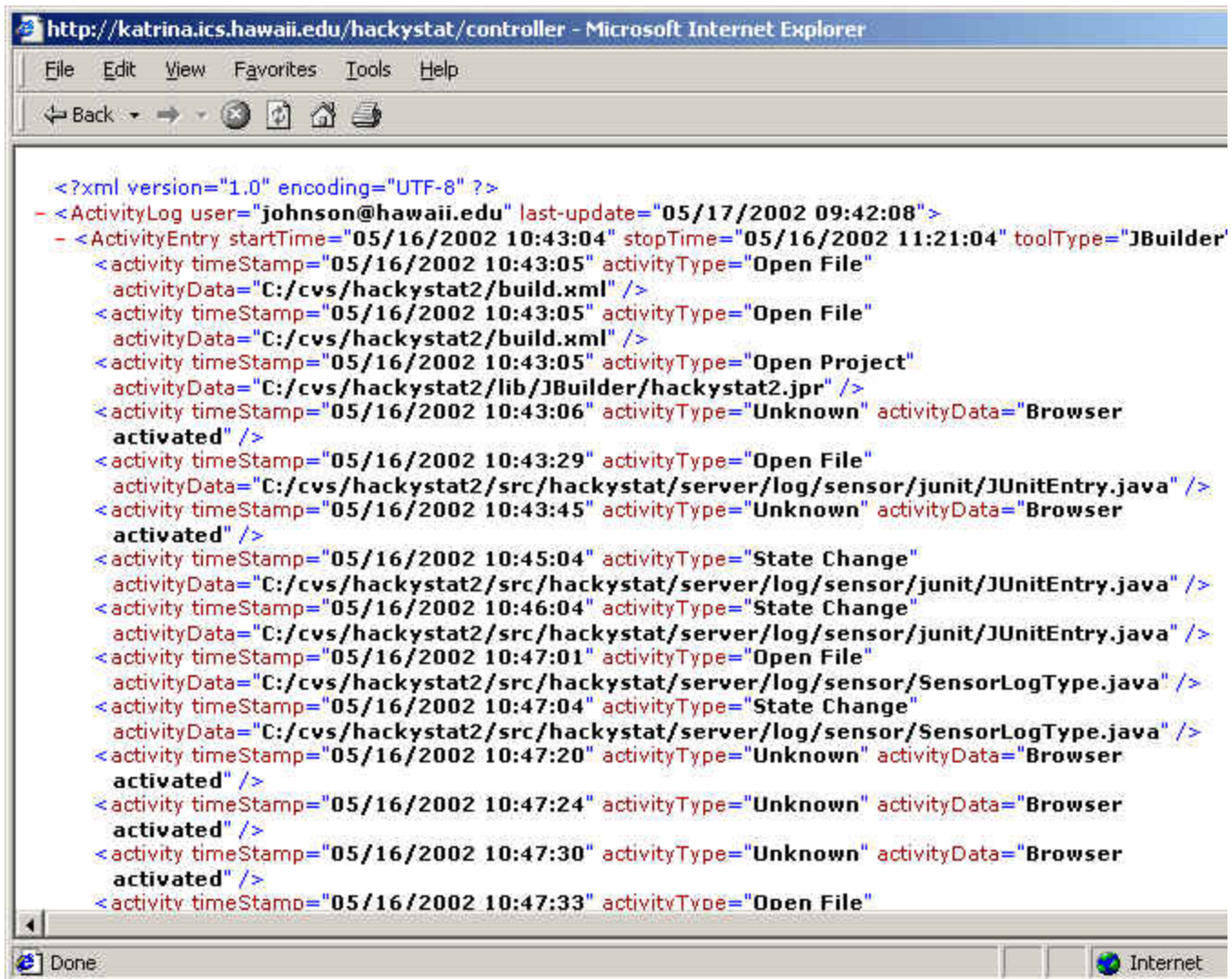
```

#### 4.6 Interoperability with other tools and data publication

While Hackystat is intended to serve as a centralized repository for certain types of software engineering data, it can also "publish" its data and analyses to other tools and repositories through its HTTP interface. For example, a tool can issue the following HTTP request to obtain raw sensor data from a Hackystat server for a particular sensor type, day, and user in XML format:

```
http://katrina.ics.hawaii.edu/hackystat/controller?CommandName=ListSensorLogDate&Key=56HDFUI432LP&Day=16&
```

Here is a screen image illustrating the XML data that would be returned by the above request if my user key had been supplied:



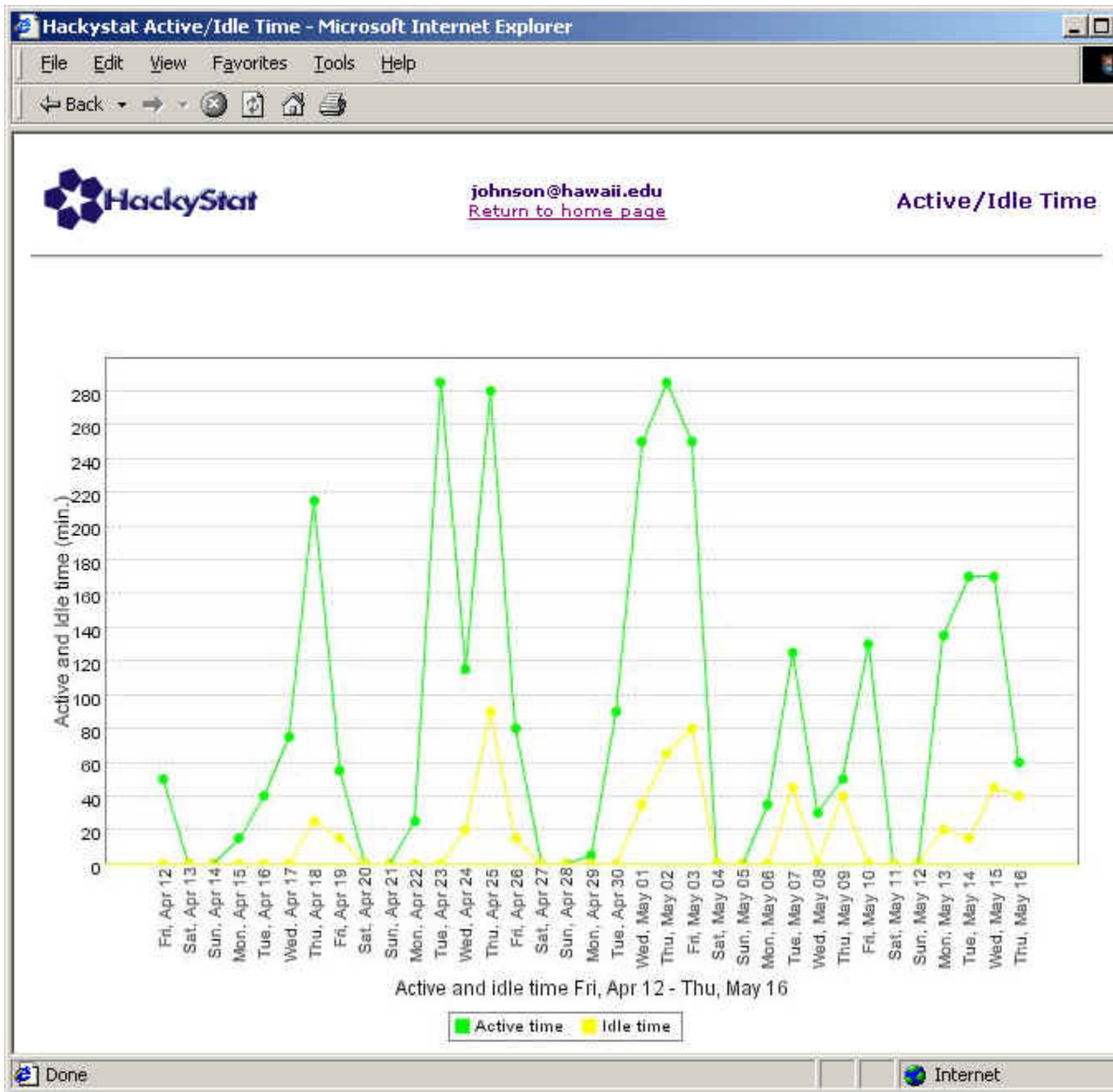
```

<?xml version="1.0" encoding="UTF-8" ?>
- <ActivityLog user="johnson@hawaii.edu" last-update="05/17/2002 09:42:08">
- <ActivityEntry startTime="05/16/2002 10:43:04" stopTime="05/16/2002 11:21:04" toolType="JBuilder"
  <activity timeStamp="05/16/2002 10:43:05" activityType="Open File"
    activityData="C:/cvs/hackystat2/build.xml" />
  <activity timeStamp="05/16/2002 10:43:05" activityType="Open File"
    activityData="C:/cvs/hackystat2/build.xml" />
  <activity timeStamp="05/16/2002 10:43:05" activityType="Open Project"
    activityData="C:/cvs/hackystat2/lib/JBuilder/hackystat2.jpr" />
  <activity timeStamp="05/16/2002 10:43:06" activityType="Unknown" activityData="Browser
    activated" />
  <activity timeStamp="05/16/2002 10:43:29" activityType="Open File"
    activityData="C:/cvs/hackystat2/src/hackystat/server/log/sensor/junit/JUnitEntry.java" />
  <activity timeStamp="05/16/2002 10:43:45" activityType="Unknown" activityData="Browser
    activated" />
  <activity timeStamp="05/16/2002 10:45:04" activityType="State Change"
    activityData="C:/cvs/hackystat2/src/hackystat/server/log/sensor/junit/JUnitEntry.java" />
  <activity timeStamp="05/16/2002 10:46:04" activityType="State Change"
    activityData="C:/cvs/hackystat2/src/hackystat/server/log/sensor/junit/JUnitEntry.java" />
  <activity timeStamp="05/16/2002 10:47:01" activityType="Open File"
    activityData="C:/cvs/hackystat2/src/hackystat/server/log/sensor/SensorLogType.java" />
  <activity timeStamp="05/16/2002 10:47:04" activityType="State Change"
    activityData="C:/cvs/hackystat2/src/hackystat/server/log/sensor/SensorLogType.java" />
  <activity timeStamp="05/16/2002 10:47:20" activityType="Unknown" activityData="Browser
    activated" />
  <activity timeStamp="05/16/2002 10:47:24" activityType="Unknown" activityData="Browser
    activated" />
  <activity timeStamp="05/16/2002 10:47:30" activityType="Unknown" activityData="Browser
    activated" />
  <activity timeStamp="05/16/2002 10:47:33" activityType="Open File"

```

#### 4.7 Graphical analyses

Finally, we have done some initial work on generating graphical representations of data. We have recently integrated an open source charting package into Hackystat and can now easily build and present charts. Here's an example chart that shows the trend in daily active and idle time:



## 5. A possible direction for joint research and development

The optimal joint research project would lead to a "win-win-win-win": improved software development at JPL/MDS, funded by HDCCSR (or some other program), and advancing the research missions of USC/CSE and UH/CSDL. The following subsections re-iterate the four requirements of the HDCCSR program and suggest ways in which we could meet them:

### 5.1 Address fundamental research issues in dependable software-based computing and communication systems,

One potential way to frame the research collaboration is in terms of addressing "cost" of dependable software development. There are actually two interpretations of "cost" that we could simultaneously address: (1) "cost" in terms of the cost of collecting metrics that are useful in developing dependable systems. Under this interpretation, what we would explore is the space of metrics that are amenable to automated collection and analysis (and thus are very low cost); (2) "cost" in terms of better prediction of the time/resources required for dependable software development. Under this interpretation, what we would explore is how to exploit the low-cost metrics we will collect to improve the calibration of Cocomo-style cost prediction model.

## 5.2 Develop research products in the form of prototype tools or methodologies,

Some of the research products we can produce include:

- A set of sensors for the JPL/MDS software development environment. In addition to the activity sensor illustrated above, my initial discussions with Carl indicate to me that the following additional sensors would be useful: (1) a "CCC/Harvest" sensor, which provides information about the set of Change Packages and their state; (2) A "Change Package" sensor, which provides the mapping between Change Package IDs and the files that are attached to them; and (3) A "Build" sensor, which provides detailed information about the build failures and the files/Change Packages that were involved, and (4) an IAR/IM sensor, which provides information on unscheduled work and the files affected.
- A set of analyses that extract and present the basic dependability metrics from the sensor data,
- A set of analyses coupled with a methodology that facilitates "root cause analysis" when dependability metrics are particularly bad (or good).
- A set of analyses coupled with a methodology for developing an "early warning system" that enables developers/managers to take corrective action regarding dependability earlier in development.
- A customized version of Cocomo that enables better cost prediction for dependable software

## 5.3 Provide dependability attributes that are suitable for measuring the impact of the research products,

I provided an initial pass at some dependability attributes above: (a) Increased Change Package throughput; (b) Improved predictability of Change Package throughput; (c) Decrease in the rate of failure during Build Queue, Build Test, and Integration Test; (d) Decrease in the average time required for "recovery" from a failure during Build Queue, Build Test, and Integration Test; (e) Increased Implementation Task Rollup throughput; and (f) Improved predictability of Implementation Task Rollups.

## 5.4 Include a plan for the empirical evaluation/validation of the proposed research products.

The empirical evaluation and validation plan might include at least the following components, each of which results in a research contribution.

- **Validation of the automatically collected metrics.** As illustrated above, the raw sensor data stream needs to be massaged to produce interpretations such as the "Most Active File", upon which higher-level analyses are built. Our plan must include validation studies to assess the accuracy of our sensors, such as the videotape study we will perform this summer. The resulting research contribution includes increased understanding of how to develop and validate metric sensors, as well as the design and implementation of a set of validated sensors.
- **Validation of the dependability attributes.** I have proposed 6 dependability attributes above. Are they the right ones? Are some missing? Are some redundant? The resulting research contribution includes better insight into the appropriate set of automatically collected metrics that yield insight into dependable software development;
- **Validation of the Cocomo-based cost prediction model.** This one should be pretty straightforward :-). I'll leave it to Dan and Barry to flesh it out.
- **Validation of the "root cause analysis" methodology.** This would involve qualitative research, in which we determine through developer interviews and surveys the extent to which the metrics data actually supported defect prevention. The resulting contribution is improved insight into the ability of these metrics to support root cause analysis and insight into future productive research directions.
- **Validation of the "early warning system".** This can involve an empirical study in which we build a developer feedback form into the emails generated by the early warning system and deploy it for a limited time. During this time, the developers would provide feedback through clicking on one of two links in the email to inform us of whether the early warning provided was either helpful or unhelpful (i.e. "false positive"). The resulting research contribution includes a better understanding of the kinds of dependability issues that are amenable to automated early warning.

In the real proposal, we'll need to organize these activities across the four-year time frame of the research project.

## 6. Issues and topics for further discussion

To conclude, here are some of the issues I've thought of while developing this white paper that I'd like to throw out for broader discussion:

- **Big Brother, developers vs. managers, and individual vs. group analyses.** So far in the design and implementation of Hackystat, we have been quite concerned with issues related to "Big Brother". For example, I have been concerned that developers in some organizations might not adopt the tool for fear that Dilbert-style managers will access their fine-grained data and use it against them. ("Jones, your unit test failure exceeded 50% this week. Get it under 25% next week or we'll have to fire you.") To



minimize that issue, we've focused on representing only individual data collection and analysis and on working toward the goal of providing helpful analyses back to developers based upon the data they provided. In other words, we have not so far developed explicit interfaces and mechanisms for "managers" or for "groups of developers". I believe that the system must provide appropriate support for managers as well as developers, and for groups as well as individuals. An interesting aspect of this project will be negotiating these differing requirements and needs successfully.

- **A detailed history of a pain package.** So far, I've not provided any detail on specific kinds of "root cause analyses" or "early warning systems" that can be provided by automated collection and analysis of sensor data for JPL/MDS. One way to flesh out the details is to obtain a better understanding of one or more "pain packages" (PP). By getting enough information about what occurred during the development of a CP that later evolved into a PP, and the kinds of activities that occurred to get the PP through the build and testing states, I believe we will be able to articulate the kinds of sensor data that, had it been present at the time, would have provided support for some form of root cause analysis and/or early warning system. This understanding will both strengthen the proposal and give us an initial design target for these capabilities in the system.
- **More details on the USC/CSE component of this collaboration.** I've not gone into detail in this white paper regarding the specific ways in which USC/CSE would collaborate (other than some general references to Cocomo-style cost estimation). I'd like to learn more about possible ways to collaborate. Dan? Barry?
- **Other funding.** Spencer Rugaber (the NSF Software Engineering Program Officer) mentioned to me that he heard that there are other program solicitations at NASA with a software engineering orientation. How do we find out more about these opportunities?
- **The testbed issue.** One potential problem with the HDCCSR program is that it talks about the need to use a "software testbed" at NASA Ames. Is it possible to include MDS software as part of this testbed? Alternatively, can we take the results of the MDS research and apply it to the NASA Ames center? Finally, I wonder if the testbed approach is even set up to accommodate the kind of in-process data collection that is being proposed in this research?.
- **Next steps for the proposal.** If we can reach agreement that a proposal seems feasible and worthwhile, I'd like us to develop a timeline for development of the proposal, and tentatively allocate roles and responsibilities.