Configuration Management and Hackystat: Initial Steps to Relating Organizational and Individual Development

Cliff Tomosada
Burt Leung
Department of Information and Computer Sciences
University of Hawai'i
Honolulu, HI 96822
tomosada@hawaii.edu
bleung@hawaii.edu

Abstract

Hackystat is a software development metrics collection tool that focuses on individual developers. Hackystat is able to provide a developer with a personal analysis of his or her unique processes. Source code configuration management (SCM) systems, on the other hand, are a means of storage for source code in a development community and serve as controller for what each individual may contribute to the community. We created a Hackystat sensor for CVS (an SCM system) in the hopes of bridging the gap between these two very different, yet related software applications. It was our hope to use the data we collected to address the issue of development conflicts that often arise in organizational development environments. We found, however, that neither application, Hackystat or CVS, could be easily reconfigured to our needs.

1. Introduction

The definition of Configuration, as determined by IEEE, is as follows:

> *"Configuration is the process of identifying and defining the items in the system, controlling the change of these items throughout their lifecycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items." [4]*

As it applies to software engineering, configuration is about the lifecycle of the software components and how code evolves over time.

Configuration Management (CM) is the process of managing the changes made to code over the lifecycle of a project [2]. It is about the dynamics of software engineering as it applies to the organizational process. CM indirectly dictates how members of a software development team work together and also how they are organized.

In this paper, we present our initial steps into bridging the gap between organizational CM and personal goals (development direction) by focusing on CM source control applications and a personal software metrics application called Hackystat.

The data collected for configuration management has a broad scope that covers all areas of the software development process. CM data for an organization is represented in their documented processes, problem report logs, repository data, project timeline, etc. We focus upon repository data for purposes of our research.

An organization following CM practices would have a central repository of information through which its members can access important company documents and project source code. Our research, however, we focus solely on source code repository data.

In a source code repository, the members of an organization (usually the developers) can access the source code for their project in a controlled manner. It is the main purpose of the repository to control the source code in such a way that it minimizes the development conflicts that arise when many people within an organization work on the same set of source files.

A source code repository supports two core functions: (1) copying of source code out of the repository and (2) adding of source code into the repository. In addition to these core functions are more complex operations such as conflict detection, which occurs when someone tries to add code that is out of date to the repository; branching, which allows developers to have their own personal copies of the source code independent of the other developers; and merging, which allows the repository to

maintain the stability of the source code. Executing these functions cause the source code within the repository to change in state. These state changes can be thought of as "events" that take place within the repository.

In general, source code repository systems are meant to control a group of developers working within the same project on a pooled code base. They provide developers with instant access to the current code base and provide means of controlling for source code conflicts. This control, however, is always enforced after-the-fact; i.e. developer conflicts only arise after the developers have completed coding and attempt to upload their code to the repository – at this point the repository informs the developer that a conflict has occurred.

At one end of the spectrum, we have CM and at the other, we have Hackystat. Hackystat works through sensors deployed on a client machine. These sensors detect events of interest and send the data associated with these events to a central server. Hackystat data, unlike CM data, focuses specifically the individual developer, rather than the entire organization.

The data collected by the Hackystat system can be viewed by going to the server's web page. The data is kept private to the user it applies to and the associated analyses that can be performed upon this data are also limited to a single user. Hackystat, therefore, cannot make determinations based upon a group of people. Hackystat data and associated analyses are helpful, however, in determining individual developer productivity and relating personal goals. For example, a developer using Hackystat could discover that he or she only codes for a total of 1 hour per week, or that he or she spends more time creating unit tests than functional code.

We have presented two types of systems so far and described their operation at a high level. Hackystat manages and analyzes personal statistics, but cannot make inferences upon how a programmer's current actions will impact others in the development organization. A source code repository system, on the other hand, knows nothing about personal statistics and cannot determine individual development goals.

One problem that arises because of this disparity between source control applications and individual developer directions is that while a developer may be aware of his or her own personal goals, he or she may not know the goals of the other developers within the organization. More importantly, a developer may not know if his or her goals conflict with the goals of others. In addition, the nature of CM source control systems make detection of conflicts possible only at the point when developers attempt to merge their individual goals together after these goals have been attained.

This "after-the-fact" conflict detection is sub-optimal with regards to software development since developers must do the work first before finding out that the work they just completed is not compatible with the current state of the system. In this case, developers not only waste time performing unnecessarily coding, they must also spend time undoing what's already been done.

By creating a Hackystat sensor to detect SCM events in real time and using the data collected from this sensor in conjunction with other Hackystat sensor data, we believe that conflicts can be effectively reduced during development, thereby increasing developer efficiency; ultimately leading to more research in relating CM organizational metrics to Hackystat analyses.

In the following sections, we discuss similar CM products along with the pros and cons of each, other research conducted on source code repositories and Hackystat systems, and development successes and failures. We conclude by presenting directions for future research and development based upon our experiences so far.

2. Related Work

In order to see what other products are out there, we looked at several commercial products. Specifically, we wanted to see what other group level analysis were offered. While there are a plethora of commercial CM systems out there, one of the most famous and highly lauded ones is Together/J. This system won Software Development Magazine's Jolt Award (May 1999), JavaWorld's Best App Award, and JARS Top 1% Award [5]. Together/J is considered by many to be the best product for design and management of Java software projects. Since it has proved itself as a top contender as a commercial CM system, we will take an in-depth look at it below. It is assumed that other CM systems will have similar and/or lesser features than that offered by Together/J.

Together/J is a company that proclaims they "create software that enables enterprises to deliver high-quality applications quickly and on-budget." It has products for development teams (e.g. ControlCenter) as well as for the individual developer (e.g. Solo). The cost of this product is quite steep, about $3500-4000. For that amount of money one gets a comprehensive development suite.

One feature it offers is quality assurance management via a testing framework that performs functional tests, nonfunctional tests and unit tests utilizing the JUnit library. The basic way it works is by taking the inputs and outputs of the component being examined and then matching these up against acceptable values. The aim of the suite was to fully automate testing. This includes automating the setup of unit tests via design patterns and functional tests to validate requirements, the running and re-running of tests, and allowing creation and use of scripts to pilot testing. Finally, results of tests can be viewed via activity diagrams.

The unit test setup feature is especially nice in that Together/J's framework uses design patterns for determining areas where tests should be implemented. For example, there is a 'PrivateTestCase' pattern that determines if any class has no test case/suite associated with it. In such a case, one will be generated automatically. The framework also has specified heuristics for organizing test cases. For example, the tests for classes in a package called com.tzo.websynergy would be found in test.com.tzo.websynergy.

Together/J takes into account that comprehensive testing of all software components in a project is impossible. In order to determine how much Unit testing is enough it offers metrics and audits that can be used to determine delimiters. For example, metrics that can indicate the current difficulty of maintaining code can be used to determine if there is too much test code to maintain. The Halstead effort metric is one measure that can give such an indication. Size metrics can also be used to determine if the limited number of unit tests is inadequate to provide complete coverage of the large amount of code it tests. An example of this would be lines of code implemented.

In order to make sure that software projects are meeting the standards set forth by management, Together/J also has an auditing tool. Such standards can be coding standards, documentation standards, etc. For example, the permission to override private methods and/or the permission to use static attributes for initialization are both coding standards that can be checked by the tool. The tool goes through all the code of a software project, records, and finally reports all violations of the standards defined. The auditing feature can be extended should the default audits not cover what an organization needs. While the default audits that come with Together/J are comprehensive, custom audits can be implemented and integrated quite easily.

Making sure that documentation and code are in-sync at all times is a very important process that can be painstaking in a large software project that employs many developers. Together/J makes this process easier by providing a feature it calls 'hyperlinking.' There are different types of hyperlinks for different purposes. For example, two use cases of a software model can be linked together to associate a semantic relationship. This is also seen in linking a UML model and its associated code together. On the other hand, separate sections of documentation may be linked together to provide for ease of navigation.

Semantic hyperlinks provide developers with a means to delineate *realization* of hierarchical levels of diagrams. For example, selecting one diagram will lead to another diagram(s) that is at a lower level of detail. The diagrams will eventually lead to the code itself. Navigation hyperlinks make any software project less intimidating by providing a convenient way for developers to find their way through documentation. Such documentation can also be diagrams that are linked together. In current or future versions of Together/J, the goal is to automate changes to code as the associated documentation is changed as well.

Currently, the only patterns that Together/J can recognize are the ones meant for unit testing mentioned previously. These types of pattern recognition are simple to implement since it involves little more than matching up code strings. However, Together/J cannot recognize software design patterns. Rather, it provides functionality that makes it easier for the developer to document them. Obviously, the effectiveness of such a feature would be similar to technologies like voice recognition in which there will probably be a lesser or greater degree of error [6].

Additional features of Together/J include: generation of UML diagrams, profiling of program execution, distributed testing, refactoring support, forward and reverse engineering of code. Documentation is also made easier since Together/J can automatically generate a JavaDocs or UML for existing code.

Another producer of CM systems is Rational, which is well known for their comprehensive software engineering products, e.g. Rational Rose. They offer two products for configuration management, ClearCase and ClearQuest. These tools provide management of the workspace, building of the code, version control, visualization of the relationships between code components, and identification of defects and changes to code. These are all features covered in Together/J's product we just described.

To date, all configuration management software out there provide a shallow analyses of data captured from the developer. Current CM systems keep a database of CM data collected and thus a history is generated for evaluation. However, there offer no deep critical analyses that would propose suggestions for change about software development within group dynamics.

While current commercial CM systems will give metrics such as total time spent on a project, lines of code, or cohesion or coupling, these features are commonplace and offer no in-depth analyses of the interactions among developers on the same team. Such commercial systems are almost always expensive as well, costing several thousands of dollars to purchase. Our research deals with analyses of group level dynamics. It involves recognizing code that is touched by an individual developer on the team and then deriving a suggestion based upon the analysis of the collective data – specifically, the ability to recognize patterns in the organization using a specific CM application that will lead to source code conflicts.

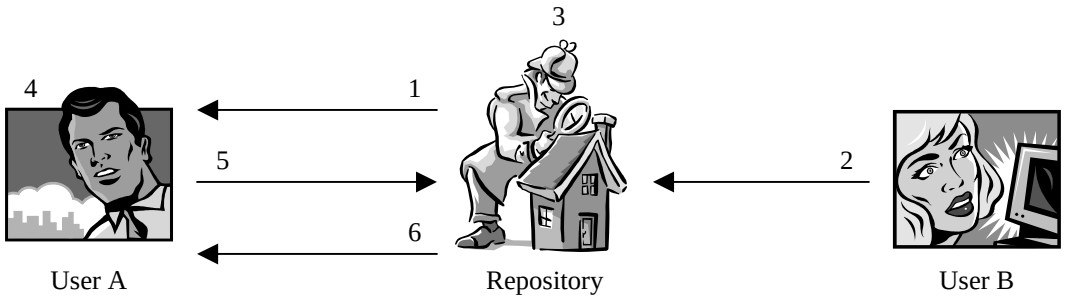

Figure 1: An illustration of events leading to a conflict (See Table 1).

While there are no commercial systems that offer this functionality, much research has been done in the domain of detecting and resolving conflicts within a development organization.

Ruiqiang Zhuang in his master's thesis proposes a method of using a web based distributed design system and intelligent agents to detect conflicts [1] – a system that seemed very similar to Hackystat. This research, however, depended upon a preconceived knowledge of the exact system to be developed. In addition, this system was meant to detect only run-time conflicts.

Leonhardt, et al. describe how they use decentralized process models that recognize patterns against historical development traits to maintain software design consistency [8]. They use a framework called ViewPoints in their research to define parts of the development process. In that respect, they are attempting to detect conflicts in software architecture. Their research deals with the high levels of design rather than development of source code.

Steve Berczuk, in his paper, Configuration Management Patterns, proposes the idea of minimizing conflict through the definition of a private and public area of source code [9]. He proposes the idea of Private Versioning as a means of backing out when a conflict occurs. He states that there should be a private repository for each developer, and a public repository among all developers. As developers make changes, they update their private repository and occasionally merge with the public repository. This way, he states, there is always a means of backing out of a conflict when it occurs. He does not focus on detecting conflicts, but rather on resolving them.

Much research has been done in the area of conflict detection and resolution. Our research in unique in that we are proposing a method of detecting conflicts between developers at code-time working in an environment that uses a standard source control application. In the next section, we explain our methodology in creating a Hackystat sensor for a source control application and the problems we encountered.

#	Event
1	User A retrieves source code version 1 from the repository.
2	User B uploads her new source code to the repository.
3	The repository source version is now 2.
4	User A works on the source code on his computer (this is the old version).
5	User A uploads his source to the repository
6	The repository informs him that his code is in conflict with the current version.

Table 1: Describing events that take place in the occurrence of conflicts.

3. Sensor Design

One of our initial goals was to create a means for Hackystat to identify occasions when a user's current development intensions were in conflict with the development intensions of others working within the same source code repository. We realized early on that the detection of this type of event would require knowledge of the developer that went outside the scope of Hackystat or any SCM application since it would require that we know the intensions (and the changes in these intensions over time) for each developer in an organization. Therefore, we focused on a simpler but related problem: detecting upcoming conflicts between developers. In Figure 1, we illustrate the progression of events that would lead to developer conflict from a conceptual standpoint. Ideally, the detection of an upcoming conflict would take place somewhere between steps 3 and 4 of Table 1, when User A's version of the source code became out of date due to User B's upload of new source code. This problem could be easily dealt with

using the data already being collected by Hackystat and by harvesting data from a source code repository.

Hackystat is a system designed to sense developer activities and send the activity data to a central server for processing. The main components of this operation are the sensors that reside on the client machine. Sensors exist for IDE's like Emacs and JBuilder; and also for project build utilities like Ant and JUnit. The sensor data contains information that indicates the files that a developer works on over a period of time.

On the other hand, SCM systems contain information about contributions to the repository over time. By collecting this information from the repository and relating it to Hackystat data, it was possible to detect upcoming development conflicts. In Table 1 above, steps 1, 2, and 3 would be information collected from the repository. Upon the occurrence of step 4, which would be detected by Hackystat, a possible upcoming conflict would be detected. Implementation of this concept, however, would not be a trivial task. In this section, we discuss how we went about implementing the data collection for SCM systems and the problems we encountered with this task.

We first had to choose the source code configuration management system we were going to use. This would be the system we would develop the Hackystat sensor for. We wanted a system that was free, open source, and widely used. Some options, like RCS and Microsoft™ Visual Source Safe, were considered but discarded because they did not meet these desires. CVS was the most logical choice because it met our requirements. An added benefit to using CVS was that it was also the system we were going to use to control our source code. Designing a Hackystat sensor for CVS would prove to be difficult, however.

Development was considered keeping three implementation objectives in mind. The first of these objectives was to create the sensor with minimal or no dependencies upon the SCM system itself (in this case, CVS). We did not want to alter the system's source code or base our sensor on any volatile part of system that may change over time. Another design objective was for the sensor to be easy for users to install and set up. We did not want to involve the user in a complex setup and maintenance routine. The last objective was to code the sensor entirely in Java. Since Hackystat was also coded in Java, it follows suit that any extension of Hackystat should also be coded in Java, save for extreme cases.

We also needed to research ways that we could detect CVS events. CVS events, in this case, were defined as any action that causes a connection to be made between the CVS client and the CVS server. In this case, a CVS "event" could be a CVS update (copying source code out) or a CVS commit (adding source code in). Unfortunately, our research led us to believe that we would not be able to detect all CVS events without violating our three design objectives. We could, through use of a built in hook function in CVS, reliably detect commit events, however, CVS did not provide a framework for plug-in development, nor did it allow for any indirect extension to its functionality. To gain access to this full range of events, we would need to alter the CVS source code (written in C), and therefore violate one of our implementation objectives. Despite these limitations, we moved ahead and developed a Hackystat sensor that would detect CVS commit events using the built in hook function which conformed to all of our implementation objectives.

We later found, that a sub optimal design decision we made regarding the way sensor data was sent would make analysis of the data difficult. In Hackystat, each user has his or her personal store of sensor data and all Hackystat sensors are configured to address a single user per session. This design was reasonable for the situation at the time; since only one person should be using an IDE or project build utility on a single computer. In our situation, however, since a CVS server handles many users on a single machine, we needed the CVS sensor to be able to adapt to multiple users and data that applied to different users of Hackystat.

At this point we saw two options. Our first option was to ignore the issue and just send the data to a single user account on the Hackystat server. This account would presumably be publicly accessible so that any user would be able to see the sensor data. The other option was to have the CVS sensor send the data for each commit, depending on the user who performed the commit, to the appropriate Hackystat user. The latter method seemed a better design decision at the time since it would preserve users' data privacy, a prominent concern of Hackystat. We later realized, however, that this solution would separate the data into individual user accounts, destroying the organizational component that the data previously represented. In short, we again constrained ourselves to analyses upon the individual, since Hackystat analyses were not meant to run across multiple users and we would not be able to detect events that occur above the individual level. Hackystat needed to be able to access all the data across all users in order to perform this function.

In addition, the sensor we developed proved to be inadequate. It did not collect enough data to allow us to conduct further research in our theory. We implemented the CVS sensor knowing we would only detect CVS commits. Only detecting commit activity, however, did not provide enough information to detect upcoming conflicts that occur at the organizational level. A critical piece of data we were missing was the detection of the CVS "update" event. With respect to Table 1, a CVS "update" is analogous to step 1, when user A downloads the source code to his computer. Without this

information, we had no idea when a particular Hackystat user updated his or her source code. Therefore, we had no idea of knowing what version of the code was on a user's machine at any given time, or at what time that version of the source became out of date. This information was needed in order to detect upcoming conflicts.

In spite of this problem, we used the commit data we were able to collect from CVS and began implementing a set of analyses that would operate upon this data. We could not detect emerging conflicts yet, but we could start implementing the framework for doing so in the future. In the following section, we discuss the analyses we implemented using the data we had available.

4. Analysis Design

Hackystat analyses are implemented as a set of web service commands on the Hackystat server. A user goes to the Hackystat homepage and requests that a certain analysis be performed on a set of his or her personal sensor data. The web service then processes the request, performs the analysis, and displays the results to the user. The most basic of these functions is a user request to view the sensor data. Some other implemented examples include active file and development curve analyses. The active file analysis operates upon the activity log data and shows the user what files were active at a certain time. The development curve analysis operates upon the OOSize and activity data to plot a graph of the development of files over time. In this section, we discuss the Hackystat web service pages and respective analyses we developed, current problems with the implementation, and future plans regarding these analyses.

Within the Hackystat server, we first created several pages that allowed the user to view the collected sensor data. The first such page displayed the sensor data in a view similar to a Windows™ Explorer file hierarchy. This page allowed the user to traverse through the different directories and files represented in the sensor logs. Each directory and file was represented as a hyperlink; clicking on a file link allowed the user to view the CVS commit actions for each file. We felt this functionality was necessary in the case of our sensor data since, at the time, other Hackystat data was displayed chronologically. We believed that displaying CVS sensor data in this manner (in chronological order rather than in file order) would be conceptually difficult since a CVS server is more representative of a repository of files rather than a timeline of events/transactions.

Hackystat, however, not only allows users to view the sensor data, it also offers specific analyses that can operate upon the collected data. These pages we created for our sensor data assisted the user in viewing the data, but did not offer any sort of helpful analysis upon the data. Our next step, then, was to create a set of pages that would display analysis results from the data we collected from the CVS server. Even though we were unable to collect the data we needed from CVS to detect upcoming conflicts, we still wanted to lay down a base set of analyses from which a conflict analysis could be derived at a later time when more data became available.

Currently, all Hackystat sensor data is based upon a set of files being edited, active, etc. We realized that our sensor data was based upon a related but different set of files. The files we dealt with were in a SCM repository, rather than on a user's computer. In the same way a user has a set of active files a CVS server would also have a set of active files. We created a "CVS active file" analysis that determined the most and least active CVS files by comparing the total number of commits along with total lines changed at each commit. Used in conjunction with the active file analysis, we believed that this analysis would assist users in relating coding effort (files that were worked on the most/least) with contribution effort in CVS (files that were committed the most/least). Coupled with other CVS event data, this analysis could provide a more accurate picture of the active CVS files.

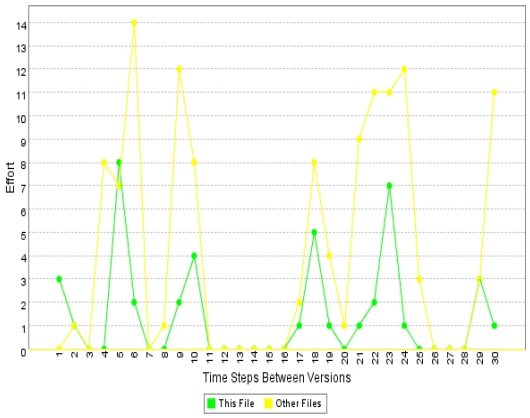

Figure 2: A graph showing the work effort analysis between two commits of a file.

We also knew that work effort could be divided into the intervals between commits of a file. We created an analysis page that allowed Hackystat users to see a graph of their work effort on a file between two commits of that file. The analysis also overlaid a graph of the combined effort on other files within that time period as well. Figure 2 shows an example graph. The graph's X-axis represents the time span between the two commits divided into arbitrary "intervals" and the Y-axis represents the collective effort as a heuristic number

analogous to a "credit". We believed that this analysis would allow users to relate their coding effort on a particular file with coding effort on other files within a specific time period. In this case, within two commits of a file.

Using the Hackystat sensor we created for detecting CVS commit events, we were able to come up with a set of analyses that operated upon the data we collected. The analyses we developed were not useful in determining upcoming developer conflicts, but instead, laid down a basic set of analyses that could be extended to detect conflicts once we were able to collect more data from CVS or some other SCM system. Even with this data, however, we would still need to be able to extend the functionality of Hackystat in such a way that analyses upon the CVS sensor data could be applied across multiple users. We discuss ways of doing this in later sections.

In the future, the CVS active file analysis could allow a Hackystat user to determine the appropriate time to commit a file by comparing the CVS update and commit events against the Hackystat sensor logs that determine file activity. Once total effort on a file reached a certain threshold, Hackystat could recommend that the user commit the file. In addition, this analysis would be useful in also informing the user about files that are not in the repository. These are files that have never been committed to the repository, but are present in the user's computer.

Similarly, the work effort analysis we developed could also be expanded in the future, using other CVS event data and data from other users in Hackystat, to determine points where development conflicts may occur. The effort graph could be overlaid with CVS event information. When certain events occur, in this case, a CVS commit by another user of the file in question, Hackystat could inform the other users editing this file that their version of the code is out of date. Optimistically, we could detect the upcoming conflict even earlier; when we notice that two users were working on the same file and that their version of the file was the most recent in the repository, we could inform both users that they are working on the same file and are heading for a potential conflict.

Analysis pages are only an initial step in extending the functionality of Hackystat. It is expected that, in the future, Hackystat user analyses will be performed automatically as data reaches the Hackystat server. When Hackystat "sees" an interesting bit of information, it would send an "alert" to the user informing them that it had seen something that may be of interest. At that point, the user has the option of taking action or ignoring the alert. In the next section, we discuss possible Hackystat alerts for detecting upcoming developer conflicts.

5. Alert Design

Unlike a Hackystat analysis, which is initiated by the Hackystat user from his or her user page, a Hackystat alert is generated automatically when Hackystat detects an interesting event. In the future, it is expected that Hackystat analyses will give way to Hackystat alerts, thus eliminating the need for a user accessible web page. A user would no longer need to visit his or her user page to analyze the sensor data: the Hackystat server would do analysis automatically and inform the user when it sees something that may be of interest.

Several Hackystat alerts already exist. They deal with things such as bad sensor data and class complexity. Hackystat users are informed through a personal email when the alerts fire. These emails are sent once per day in the early morning hours. As such, there could be up to a 24-hour delay between the time an interesting event occurs and the time a user is informed about that event. Whether or not this is a suitable solution, however, is not the focus of our research.

We did not create any Hackystat alerts in this iteration of development, since we did not believe that we had an analysis that would provide any interesting information to the user warranting the creation of an alert. However, we did have an unrealized concept of how to detect upcoming conflicts. Development of this idea would warrant the creation of a Hackystat alert. In this section, we discuss how such an alert could be developed in Hackystat.

Essentially, all Hackystat sensor data can be characterized as a stream of events that arrive at the Hackystat server in chronological order. Hackystat analyses operate on this stream, but only do so upon a user request. By creating an alert-style function, the data can be analyzed automatically; either in real time as it arrives on the server, or once a day as is done now.

Assuming that we are able to collect the data we require from the CM source repository, we expect a Hackystat alert for upcoming conflicts to operate by detecting events depicted in steps 1 – 4 of Table 1. In keeping with the current set of alerts in Hackystat, we would have a daily analysis function that would be run automatically each morning upon the data received from the previous day. As the analysis iterates through the list of sensor data, it would append events of interest into an event queue. Events of interest, in this case, include CM repository events and files undergoing active development.

This event queue would hold a collection of events that occurred across the users of Hackystat for that day. The queue would then be traversed according to a grammar or other pattern-matching algorithm. Any matches would indicate an upcoming conflict between developers, and those developers could be notified.

Following is a simple BNF-type grammar that describes a pattern in the event stream indicating an upcoming developer conflict. In this example, let E = any event in the queue, $U_{n, x}$ = an update event for a user n upon file x, $C_{n, x}$ = a commit event for a user n upon file x, and $A_{n, x}$ = any activity event with respect to a user n that changes the file x in question:

$$[E]^* [U_{n, x}]+ [E]^* (Q \,||\, Y) [E]^*$$

Where: $Q = [C_{j, x}]+ [E]^* [A_{n, x}]+$ and
$Y = [A_{n, x}]+ [E]^* [C_{j, x}]+$

Basically, the grammar describes a sequence of events that apply to the same file such that, aside from irrelevant events represented by E, if a user does an update, and if that user edits the file, and if another user at some point commits a new version of the file to the repository, a possible conflict has been detected. Note that this event stream can be related back to Table 1.

After a pattern match has been made, an appropriate alert could be sent to the affected users indicating that they are headed for a conflict in the future, when they will commit the affected file(s) back to the repository.

Of course, none of this functionality has been implemented. We are waiting until we are able to collect the data required to perform such an analysis. In this section, we described one possible way of implementing a Hackystat alert that would allow users to be informed when an upcoming conflict has been detected. In the next section, we expand upon directions for future development and research based upon what we have learned so far.

6. Future Directions

As stated before in this paper, we were unable to collect the data we needed from the SCM system we used (CVS). Without this data, we were left unable to further our research into using Hackystat along with CM data to detect upcoming developer conflicts. In this section, we discuss this problem, other problems, alternative routes we considered during the development process, and finally conclude future research directions.

Because of the many limitations of CVS, we began preliminary research into another similar configuration management software package, Subversion. Subversion is another free, open source system. It is currently under development and boasts newer and better functionality than CVS [3]. However, after reading the documentation and submitting an inquiry to the Subversion development team, we found that although Subversion had many hook functions to allow detection of various CM events, no such mechanism existed to detect "update" events. This was a critical piece of data we needed to be able to detect developer conflict.

One of our implementation objectives was to create the sensor with minimal dependency upon the tool (in this case, CVS). This objective limited the ways in which we could collect data from the tool we chose. Altering CVS source code was undesirable, since it would require users of the CVS sensor to download and install our customized version of CVS. It may be, however, that in order to collect the data we need, we may need to alter the CVS source code in order to add in a hook function that will allow the CVS sensor to detect update events.

There may be another option, however. CVS operates through use of two "cvs" executable files, one located on the client, the other on the server. When a user invokes CVS, the two executables make a connection and transfer information. We may want to look into creating our own simple cvs executable that would wrap the real cvs executable. In this case, our executable, when invoked, would call the CVS sensor, and then invoke the real cvs executable (that had been previously renamed or relocated). This would simplify the install process such that a user would only have to download our cvs wrapper executable to the right directory, and rename the real cvs executable to another name, like "real-cvs". Conceptually, the idea seems simple, however, not enough research has been done into the cvs executable application to determine if this is a feasible course for development.

Another difficulty we faced was the problem of data storing and analysis. We made a decision to store the sensor data in such a way that the user's privacy was kept. This limited our analysis ability, because Hackystat was not designed to perform analysis across users. Since we still would like data to remain private, one option that would allow for multi-user analysis, while preserving data privacy, would be to modify Hackystat to allow individual users to select a group (in this case, a CVS server) to which they belong. Hackystat would store a mapping of user to CVS servers. When a user requests an analysis for CVS sensor data, the analysis command would traverse the user list and use only the data from those users whose CVS servers match that of the user requesting the analysis. In this way, we preserve the individual's data privacy while allowing data analysis across multiple users.

The preceding paragraphs enumerated several future development considerations. In the remainder of this section, we discuss other research considerations that should both precede and succeed this phase of development.

Due to the time constraints of this research project, we were unable conduct research in evaluating the problem. In this case, the problem was that of developer conflicts leading to decreased productivity and increased system

defects. We know that development conflicts arise regularly within a software development organization, but we do not really know the impact that these conflicts have on developer productivity. Ideally, research should be conducted using direct observation, retrospective interview, and bug tracking data to determine, if in fact, occurrences of developer conflicts contribute to the reduction of overall productivity and if this is related to increased system defects.

It is hard to quantify the number of future research opportunities into this domain. Many other types of CM applications exist from which data can be harvested. The data, such as that from a bug tracker application, could be used in determining why defects occur in a system. In other words, it is inevitable that a developer will introduce bugs into a software system, so exactly what is it about the actions of a developer that make him or her directly responsible for introducing a bug into the system? This is a very hard question to answer, but research has already begun on this problem [7].

Another, perhaps easier problem dealing with bug tracking data deals with developer activity. Is there a relationship between defects and developer activity? In other words, does a high level of developer activity correlate with increased system defects? If so, a can a Hackystat alert be created to inform the developer to relax or work harder; like some sort of "you-need-to-take-a-break" and "you're-not-working-enough" alert?

Relating the organization to the individual is a broad topic that covers a wide range of applications. Much of this is undiscovered territory. This phase of research and development covers only a fraction of what is out there to be discovered. The sky is definitely not the limit.

7. Acknowledgements

We would like to thank Dr. Philip Johnson for guiding us in our research and providing insight into our methodology. In addition we would like to acknowledge the other students our ICS 691 class who have contributed their ideas and suggestions, supporting our work.

8. References

[1] Ruiqiang Zhuang. Conflict Detection in Web Based Concurrent Engineering Design. Masters Thesis Proposal, University of Florida, 1999.

[2] Configuration Management Inc. Overview of software configuration management. Online documentation: http://www.softwareconfiguration.com/About%20CM/aboutCM_overview_layers.htm, 1999.

[3] Subversion. Project home site for Subversion version control software: http://subversion.tigris.org/, 2002.

[4] IEEE. The configuration management definition. Online documentation: http://www.lal.in2p3.fr/~arnault/Atlas/cern-feb-99/tsld003.htm.

[5] Frank Baker. Together/J Wins JOLT Award. http://www.multitask.com.au/JavaNews, May 13, 1999.

[6] Together/J. Together/J Developer's Guide, Chapter 6, pp.111-161, 2002.

[7] Koji Torii, et al. Ginger2: An Environment for Computer-Aided Empirical Software Engineering. Retrieved from the World Vide Web. IEEE Transactions on Software Engineering, Vol 25, No. 4, July/August 1999.

[8] Ulf Leonhardt, et al. Decentralised Process Enactment in a Multi-Perspective Development Environment. Imperial College & City University, London, 1995.

[9] Steve Berczuk. Configuration Management Patterns. Optimax Systems Corporation, Cambridge, MA, 1996.