Jacob Hyman
6.191

Proposed Research Topic

The bulk of this term in 6.191 has been spent examining various technologies currently being researched at MIT for the purpose of developing a Meng thesis project. Outside of my major in computer science, I have an interest in psychology, so I was looking for technology that could be used in demonstrating how people think about or interact with computers. After an initial exploration of research groups at MIT, I was interested in exploring a topic relevant to the research the Sociable Media Group (Media Lab) is doing. However, a meeting with the group revealed the technology they were investigating was not what I expected. Additionally, the head of the group did not seem very excited about picking up a Meng student at this time. Pursuing things further seemed to benefit neither party.

Further exploration lead to the area in which I am now focusing. The dynamic invariant detection being worked on by Prof. Ernst in the LCS Software Design Group presents an interesting computer science problem with the plenty of room for future research. Most importantly, there is a lot of potential for demonstrating a way in which this technology can improve a programmer's ability to create or modify code.

To understand the primary technology involved in Daikon (the program that dynamically detects invariants), a reader must first know what an invariant is. Simply put, an invariant is any property that holds true during various points of program execution. For example, if three variables a, b, c are always related by the equation $c^2 = a^2 + b^2$ during every execution of a loop, $c^2 = a^2 + b^2$ is an invariant for that loop. Examining what kinds of invariants Daikon can detect will help answer the question of why these invariants (and Daikon) are useful. Consequently, the following description of Daikon will be more concerned with what it does, and only briefly describe the general strategy used to accomplish the invariant detection.

A brief overview of how Daikon detects invariants will help explain some of the limitations of dynamic invariant detection. The most important fact to note is that the

dynamic detection of invariants relies on both a program and its input. Daikon generates possible invariants by looking at the values of program variables during important points of execution. The values of these variables are usually dependent on input, so the quality of invariant detection is a function of input. Input that presents a wide variety of values to a given variable will lead to detection of more meaningful invariants, while one run of the program on a simple input set will result in only a few generally poor invariants.

Knowing that Daikon's detection ability is decided by what input a program is run over, it is logical to examine what types of invariants Daikon looks for. The first class of invariants that Daikon tries to recognize is simple variable invariants. An invariant that holds over any single variable (e.g. variable a is never assigned to null) falls into this classification. Numeric invariants among up to 3 variables (e.g. $a < 5$ or $a + b < c$) also fall into this categorization. The last types of invariants in this classification are invariants over sequences and those involving a sequence and a number. These features allow Daikon to detect whether or not a sequence is sorted or whether or not a sequence contains a certain value.

Diakon also has the ability to detect invariants about code objects. These types of invariants are sometimes referred to as representation or class invariants and are studied in great detail during 6.170. These invariants can help other programmers treat objects like black boxes to build code around because they can be used to describe pre-conditions and post-conditions for an object's methods. What this means is that if the input meets certain specifications, the output will always meet certain specifications. Object invariants are also useful to a programmer because they can be used to determine whether their code for the representation is correct. For example, if a programmer designs a class that relies on the assumption that a string is always null-terminated, he can run Daikon and see if this invariant holds. If the invariant does not hold, the programmer has made an error in assuming strings will always be null-terminated and his code may be incorrect.

In addition to invariants over variables explicitly declared in code, Daikon has the ability to detect derived variable invariants. Using variables declared in the code, the

program derives implicit variables. These implicit variables can be used in new invariant relationships. Listing what types of derived invariants Daikon can detect will make this process clearer. Daikon can detect derived invariants over any sequence (e.g. the length of a sequence < 10), a numeric sequence (e.g. the minimum value in a sequence is 0), and a sequence and numeric value (e.g. properties of subsequences). Additionally, Daikon has the ability to determine information about function calls. In determining derived variables Daikon has to be careful because derived variables can lead to further derived variables and Daikon could theoretically derive new variables forever. This problem is solved by placing a user-defined limit on how many iterations to use when determining new derived variables.

With all these possible invariants to look for, Daikon needs strategies to eliminate invariants that are not of use to a programmer. The first step in this elimination is that the program is designed to only look for invariants that are described as "basic, general, and useful" by Prof. Ernst. Prof. Ernst has developed a sense for what invariants are useful over the course of his career and it is beyond the scope of this paper to explain this notion in any more detail. Additionally, Daikon considers the comparability of variables before deciding to see if relations hold between them. Comparing an integer's and a string's size does not make much sense. After restricting the invariants looked for, Daikon also has methods to eliminate invariants that it determines are useless during detection. If the existence of one invariant implies the existence of another invariant, only the first invariant needs to be tested for. The last step in invariant elimination involves testing how statistically valid an invariant is. Relationships that only appear once during all executions of code are probably not meaningful and possibly not even correct. To deal with this problem, Daikon allows the user to set parameters on how statistically relevant an invariant needs to be before it is reported.

This paper has not presented a comprehensive list of the types of invariants that Daikon can detect, but it has outlined the majority of invariants. From this information, the reader should be able to develop a good idea of what Daikon is capable of doing. This idea of what Daikon can do leads into the question of why invariant detection is

useful. One possible use of this tool is to recover formal specifications from code that has none provided by the programmer. Formal specifications describe how to use a piece of code as a black box. When a programmer tries to incorporate someone else's code into their design, all too often there are very limited specifications described for how the code should behave. The ability to recover undocumented specifications allows a programmer to utilize another person's code with more ease and confidence. The same advantages hold true for using any undocumented code. Program invariants can allow a programmer to see how data structures are organized and even how the original programmer envisioned the code working. Interestingly enough, Daikon's detection of invariants allows the latter to hold true even if the programmer did not have a conscious idea of his design strategy. Also, invariants can help the original programmer in finding bugs, determining if changes made to the code had unexpected effects, and even determining if a test suite is comprehensive in its scope. There have been several small studies on various code (examples from students, textbooks, and test suites) that show that Daikon does help in the mentioned areas. However, the existing data is very qualitative in nature which may not convince people of Daikon's usefulness.

My goal in demonstrating this technology is to show that Daikon can be a very useful tool to help increase a programmer's productivity. Prof. Ernst has listed several areas in which he would like to see research continue on Daikon. I feel that a few of these areas could lead to information which can be used to demonstrate how useful Daikon can be to programmers. The first area is simply more user studies. As mentioned before, I have a strong interest in psychology and am very curious as to how a tool like Daikon can change a programmer's strategy when designing code. After meeting with Prof. Ernst, it is very apparent that he feels more user studies would greatly improve the ability to explain the usefulness of Daikon. Additionally, user studies would help determine what types of invariants are most useful to programmers and lead to adding new invariants for Daikon to detect. For example, Prof. Ernst is interested in exploring temporal invariants (something holds at a certain time) and conditional invariants. Implementation of these invariants along with testing could be very useful in

demonstrating Daikon's potential. The final area of future study involves the interface with which a user interacts with Daikon. Invariants are only useful if the programmer can use them efficiently. A better interface will allow the user to intuitively select what types of invariants are important to their task. The detected invariants can then be presented to the user in a manner that is best suited to their current programming task. There is even potential for AI development in determining what invariants to report to the user.

The end goal of this research is to demonstrate Daikon as a marketable tool and clearly show how it can improve a programmer's efficiency. The package would probably include a suite of tests that show Daikon's ability, statistically significant data from user studies, and a demonstration of the improved interface. The package should allow a programmer to sit down and use Daikon to help them create or modify code almost immediately.

My proposed plan is not without potential difficulties. At this point I do not have a position in Prof. Ernst's group. We have discussed my interests and ideas for the project, and both of us are deciding whether or not it would be mutually beneficial to pursue these ideas further. The other main obstacle is the difficulty in conducting usability studies for a tool like Daikon. A study that would reveal quantifiable data may be too difficult to complete given the nature of the tool. However, in the course of working with Daikon, other areas of study will undoubtedly reveal themselves.

In conclusion, I present a schedule to complete this project. In the spring of 2002, I hope to start working in Prof. Ernst's group as a UROP and become familiar with using Daikon. Near the end of the term and during the summer I would develop and begin carrying out a usability study. By the end of the summer I will have begun coding any GUI enhancements or new invariant detection algorithms. During the fall of 2002, I hope to finish up the studies and complete the coding I started during the summer. This leaves the spring of 2003 to write my thesis and present my work.