Often, software bugs arise from erroneous assumptions made by developers about the state of the system and the inputs presented to it. Besides the obvious types of bugs which cause a program to crash, more subtle are the bugs that produce incorrect results (and often are more costly). An invariant analysis tries to quantify the invariants of a system by analyzing the state of the system over time and building up assumptions about the variants in that system. If these assumptions, or more precisely rules, are violated, then unintended program behavior, often leading to incorrect results or crashes, can occur.

DIDUCE is an implementation of an invariant analysis system for Java. It operates on class files (no source code is required) by instrumenting them using the ByteCode Engineering Library. The instrumented files are the packaged into a JAR file, which are subsequently loaded when the program to be analysis is run.

Instrumentation is performed at several locations. First, any reads or writes to a static variable is instrumented. Object reads and writes are also instrumented. Lastly, call sites, or rather their procedure arguments and return value, are candidates for instrumentation. What is notably not instrumented are all stack (local) variables (unlike Daikon); the basic assumption is that local variables generally are not interesting in an invariant analysis (this also excludes objects, because objects are created on the heap).

For each instrumentation point, a set of rules are derived that represent the "invariants" of that point. Each rule is based off of observed properties of the instrumented expressions at that point. The following are the types of expressions that are tracked by DIDUCE: boolean, byte, char, short, int, long, and reference. The value of a numeric expression is simply its integer equivalent (although the paper fails to mention how longs, which have a size of 64 bits, map to an integer which may be size 32 bits). For reference types, an integer value is obtained by hashing the string representation of the runtime type of the reference.

Once expressions have been converted into an integer format, three values are collected for each expression. The first value is simply the value of the expression. The second value is the parent object, if the expression is a field of an object. The last expression is the difference between the old value of a variable and the new value (e.g. if X currently has the value 3, and then the expression X = 1 is evaluated, we calculate the difference between 3 and 1).

The internal representation of each collected data point is a tuple (V, M), where V is the value being stored, and M is its mask. The mask M is calculated by setting equal to 1 all bits who have consistently had the same value in V. For instance, if bit B in all observed values of V has consistently been 0, then bit B in M is set to 1. Similarly, if bit B in all values of V has consistently been 1, then bit B in M is also set to 1. However, if the value of bit B in V has not been consistent, than bit B in M is set to 0.

The values of the bits in the mask can tell us much about the characteristics of the expression. For instance, if the high order bit is 1, then the expression is consistently either positive or negative (if it is a signed value). If the low order bit is 1, then the

expression is consistently either odd or even. Lastly, if all the bits are set, then expression's value is constant.

The bits in the mask are also used in a second fashion – to calculate the confidence of rules on the expression. The confidence of the rule is defined as (# of times the expression has been seen / # of 0s in the mask). Intuitively, we can see that if the expression hasn't varied much, the number of 0s will be small, and our confidence factor will be large. When a rule violation occurs, the change in confidence factor is reported. Once an "invariant" has been violated, the rules that were violated are relaxed, to include the value that caused the violation (unlike Daikon). Large changes in confidence tend to indicate major changes, and can be used to prioritize invariant violation investigations.

Several features have been added to the DIDUCE system, to increase ease of use and to make it more efficient. First, the degree of invariant tracking is user-tunable. One example of why this might be good is to eliminate reference type invariant comparisons in a program that does not use polymorphism. Since there is a cost involved in processing each instrumentation point, by removing unnecessary instrumentation points we decrease total processing time. We can also save our sets of rules from one run to the next, which is especially useful if the program must be invoked on discrete sets of data. We can apply a threshold to confidence change levels, thereby eliminating noise from minor variances. Lastly, since each instrumentation is performed independently, DIDUCE can be parallelized to instrument portions of the expression space on multiple nodes.

Experimental results showed that DIDUCE was substantially noisy in the initial portion of a program's runtime, thereby limiting the usefulness of data obtained there. On the other hand, almost all violations after an extended period of time were found to be at least "interesting," if not necessarily programming failures. Much emphasis was also placed on DIDUCE catching invariant violations reported just before incorrect results were obtained or critical failures were experienced. One area which was not explored much was the usefulness of DIDUCE to describe invariants without a need precipitated by a bug, although this usefulness was alluded to in a few places.