An Evaluation of Daikon: A Dynamic Invariant Detector

Miryung Kim and Andrew Petersen
{miryung, petersen}@cs.washington.edu

1 Introduction

Elements of the software engineering community have found that spending time discovering and documenting invariants in a program has several advantages, and their use, while not widespread, is significant. The project page for Daikon lists several possible uses, including documenting code, verifying program correctness, and validating test suites [2]. Ernst gave a colloquium at the beginning of this quarter, and we were very impressed by his presentation. Inspired by his presentation, for a term project in an architecture class, we decided to employ invariants to verify that a processor was correctly executing a program. In our scheme, invariants are discovered and checks are inserted into the code, with the hope that they will increase confidence in proper execution. If performed manually, the discovery of invariants and their insertion into code is, at best, tedious and error-prone, so we hoped to begin to automate the process with an invariant detector. In this report, we first discuss invariants and their role in our architecture project and then document our experiences in choosing and installing an invariant detector. Next, we focus on Daikon, the invariant detector we selected, and comment on the results we obtained (or didn't obtain), discuss features we would have liked to see, and mention improvements we think are necessary before invariant checkers will enter widespread use.

2 Invariants and Invariant Checking

Invariants are program properties that are likely to hold at a certain point in the code. Often, they refer to the values that a single variable can take or document relationships between data members that hold for all possible values of the members involved. Invariants are commonly used to enforce program safety (and, as a special case, type safety). Programmers are encouraged to annotate their programs with invariants, with the aim of improving maintainability, readability, and ease of verification. Assert statements that check invariants can be used to verify the state of the program during execution, and proofs of correctness often rely on annotation that provide program invariants. The insertion of invariant checks is also helpful in evolving software, as modifications that break expected invariants can be found more easily, which prevents bugs from being released. While not used universally, invariants are commonly documented (often as pre- and post-conditions of functions), and when explicitly stated and used, are thought to make the design, implementation, and maintenance of programs easier.

Therefore, the software engineering community has exerted a great deal of effort towards building automatic invariant generators and checkers, to make the creation of invariants less manpower intensive and to encourage completeness of invariants included in programs. However, at present, most invariant generation is performed manually, and tools for automating invariant detection are limited in power and effectiveness. First, invariant detectors cannot discover a complete set of invariants, because the problem of determining all invariants is undecidable. Second, invariant generation tools usually operate at the level of functional granularity, partly because the idea for invariants developed from precondition, post condition and loop invariants, which are derived from methods that consider basic blocks, and partly because statement analysis is extremely expensive in terms of time. As a result, many invariant detectors focus on relationships between the parameters passed to the function and the return value, because they generally begin their search by looking at those values and examine only transformations performed on them. Hence, invariant detectors often fail to suggest interesting invariants that involve local data allocated in the callee's stack. Finally, substantial amounts of input from the programmer, either in the form of annotations or test cases, are required to direct the checker towards useful invariants.

3 Choosing the Right Tool

In the processor verification scheme we researched for our architecture project, if code was annotated with invariants during development, our technique did not require additional work on the part of the programmer. However, if not already part of the development process, we hoped to encourage the use of invariants in general during the design and testing process. With the facilitation of the these goals in mind, we tried to select an automatic invariant detector that uses any provided annotations while requiring a minimum of user input and could be made to automatically insert invariant checks into code.

The available tools can be roughly divided into two basic groups: static checkers with "annotation assistants" and dynamic detectors. The two groups were developed for slightly different, but related, purposes. Static analysis tools are generally used to generate and verify annotations and analyze the code for type and memory safety, while dynamic tools are primarily used to detect possible invariants. In terms of input, static invariant checkers need not run the program but must be guided by human input and as such, often require annotations or specifications created by the program designer or implementer. A dynamic detector, in contrast, usually does not require additional initial input from a human (although it will accept and use annotations if available) but must execute the program being analyzed with a large test suite to infer possible invariants. However, the dynamic detector does require that a human filter the output for spurious results [2]. Unfortunately, while the two groups differ in the amount of user input required, all current invariant detectors have several shortcomings due to the intrinsic difficulty of invariant generation, as discussed in Section 2.

3.1 Static Invariant Checking Tools (ESC Java/Houdini)

ESC/Java is a static checker for Java and extends commonly performed static analysis (type checking) by also verifying memory errors like memory bounds errors and null dereferencing. However, it does not attempt to "prove" the correctness of a program. Nevertheless, ESC/Java requires a large number of annotations (usually provided by the programmer), as it works by assuming annotations are correct and searching for counter examples. Houdini was developed to reduce the amount of manpower needed to use ESC/Java and is described as an "annotation assistant." The program designers must still provide some annotations, but Houdini augments their annotations with some commonly expected invariants (comparisons between values of various data members, array lengths, and boolean predicates), in an attempt to provide ESC/Java with the annotations it requires to prove program properties.

Together, the two programs have been shown to be quite effective, and we could have used Houdini to provide the invariants we needed to verify processor execution in our architecture project. Since the program does not have to be run, a test suite does not have to be created, and programmer intervention is not necessary to filter possible invariants (ESC/Java removes invariants that cannot be proved to hold), Houdini and ESC/Java are excellent candidates. However, Houdini does not reveal the invariants it discovers (which reduces the benefit to the programmer), so if additional annotations are required, the programmer must analyze the code, attempt to annotate the more complex properties of the functions that cannot be verified by ESC/Java and Houdini, and begin the process again.

3.2 Dynamic Invariant Detectors (Daikon)

Daikon, in contrast, is a dynamic invariant detector for Java, C, and C++. It does not perform any analysis on the program. In fact, it does not use the source code at all when inferring invariants and does not initially require any programmer-provided annotations, although it will accept guidance provided as notations in files generated when the code is instrumented. However, a large, complete test suite is required. In addition, like ESC/Java and Houdini, programmer assistance is required after the analysis has been performed and output has been provided. As some of the candidate invariants provided by Daikon are spurious – caused by flaws in the test suite provided, a programmer must filter the output and possibly enhance the test suite [3].

Of these drawbacks, the necessity of running the program being analyzed is the most serious, as it complicates the automation of the verification technique we researched. Nevertheless, we chose to use Daikon for our architecture project, as we believed it was more flexible and widely applicable. It supports two additional languages in addition to Java, only requires a test suite (which most development projects

should already be creating), and provides feedback to the programmer that can be used to improve both the test suite and the program. We believe that these advantages most directly support our goals, which were to minimize additional work by the programmer and to encourage the use of invariants in development and testing.

4 Installation

Michael Ernst provides a large amount of support for installations, and Daikon can run on a wide range of platforms. Most of the available installation instructions are contained in the Daikon user manual [2], but additional general information is available in technical papers from the website. The instructions for installing both front-ends are complete and clear, and most of the problems we encountered were explained in the troubleshooting section of the user manual.

We attempted to install Daikon, including both front-ends, on both Windows 2000 (using Cygwin) and Linux, and on the whole, the process, while not painless, was fairly straightforward. However, we did encounter (or foresee) a few problems:

- Specifying Paths:
  The Daikon package supplies two scripts that set necessary path variables. They need to be tailored to each individual system, and while this is not unexpected, it was unexpectedly difficult to do so in the Cygwin install we performed. This is an issue with Cygwin, rather than Daikon, but it should probably be addressed, since Cygwin is a critical component of the recommended method for installation in Windows.
- Compiling dfej:
  The source for dfej is available, as is a binary executable. In most cases, the binary executable is probably sufficient, but we needed to recompile it for Linux and found that changes to the source code (mainly type issues) were necessary to do so.
- Using dfec
  The source for dfec is not available. They offer to recompile for your platform if necessary, but this may not always be feasible (or easy).

In summary, all of the problems we encountered during installation were surmountable. Installing Daikon is extremely easy in comparison to other Linux/UNIX tools, and ample support is provided.

5 Using Daikon's dfec: Experiments with C and C++

Daikon has two front-ends used to instrument programs. One, dfej, instruments java programs, and the other, dfec, instruments C++ programs. Technically, however, dfec also works on C code, since C programs are syntactically legal in C++. Some differences in results can be seen when comparisons are made between Daikon output on C and C++ programs, probably due to the methodology used while coding. C programs are often functional in nature, while C++ programs more easily support object-oriented design and the short, fairly simple methods upon which Daikon works well. However, we found that working in C was actually easier than working in C++, since many problems occurred when dfec was used on code that used C++ style pass-by-reference or included overloaded operators.

Our example applications were selected to represent different categories and were written to emphasize situations we encountered in larger programs:

- Exiting Functions: Multiple vs. Single Exit Points
  - Compute "abs(X)," the absolute value of X, and return values in different points of the code. (Section 5.1.1)
- Accessing Parameters: by Reference or by Value
  - Add arrays of integers by taking arrays of integers passed by reference. (Section 5.1.2)
  - Add integers by taking single integers passed by value.
- Using Standard Library Functions
  - Solve mathematical analysis problems using programs that rely heavily on functions defined in math.h. (Section 5.1.3)

- Writing Programs that Utilize Functions vs. Programs that use Inline Statements
    - Use a function to add two integers.
    - Implement the same functionality (addition) using inline statements. (Section 5.1.4)

5.1 C

To highlight some of our experiences with Daikon and its dfec front-end, we introduce four examples that illustrate the major successes or failures we encountered. The first is an example of the general case, the second illustrates the treatment of arrays, the third concentrates on Daikon's behavior with respect to library calls, and the fourth contrasts the output obtained on functional implementations and statement centered implementations of the same algorithm.

5.1.1 A Simple Program: ABS(x)

Often, when a programmer wants to check an invariant, he or she visualizes it in the form of a simple Boolean-valued mathematical equation. As illustrated by the function "int ABS(int x)" in **Table 1** , the invariants the programmer expects are very simple and comprehensible. However, the ones generated by Daikon are somewhat complex and unexpected, so the programmer needs to spend some effort to understand and apply them in context. The effort necessary to understand them probably decreases with experience with Daikon, but the learning curve on this tool seems steeper than it needs to be.

The programmer may have difficulties understanding the meaning of the provided invariants because Daikon can't intelligently integrate invariants from different exit points in a function. In some sense, Daikon is expecting a "proper program," as Parnas defines it. Even though it does some integration, the meaning can become distorted or too general, as can be see in the table below. Note also that the invariants are sound in the sense that they are true but are broad and not as strong as they could be. To get the expected invariant, we need to use Daikon's "conditional invariant" feature. This feature often requires additional input from the programmer but yields invariants that depend on a conditional – as the programmer's expected invariant, below, does.

Code	User's Expected Invariants	Invariants Detected by Daikon Note: It does not display all the invariants detected by Daikon.

int ABS(int x)
{
if (x>0) return x;
else return (x*(-1));
}
int main ()
{
int i=0;
int abs_i;
for (i=-5000;i<5000;i++)
{
abs_i=ABS(i);
}
}
	Return value of ABS(x) ==
	(x>0) ? x: -x;
		==========
		std.ABS(int;):::ENTER
		==========
		std.ABS(int;):::EXIT1
		x == return
		==========
		std.ABS(int;):::EXIT2
		return == - x
		==========
		std.ABS(int;):::EXIT
		x == orig(x)
		x <= return
		==========

Table 1: Comparison of invariants expected by the programmer and those generated by Daikon.

5.1.2 Arrays: Vector Addition

In this example, **Table 2** reveals Daikon's predisposition towards finding the range of elements in an array of data and the boundaries of the array and the difficulty of discovering interesting relationships between the elements in the arrays.

Code	User's Expected Invariants	Invariants Detected by Daikon Note: It does not display all the invariants detected by Daikon.

void foo(int cnt, int *c, int* a, int* b){
int i;
for (i=0;i<cnt;i++){
c[i]=a[i]+b[i];
}
}
int main (){
int i,k;
int a[100]; int b[100]; int c[100];
for (k=0;k<100;k++) {
 	
for (i=0;i<100;i++){
a[i]=(int) (rand()%10);
b[i]=(int) (rand()%10);
}
foo(100,c,a,b);
}
}
	std.foo():::ENTER
	a[] elements>=0
	a[] elements<10
	b[] elements>=0
	b[] elements<10
	
	==========
	std.foo():::EXIT1
	… the same as invariants generated by Daikon.
	
	In addition,
	c[]=a[]+b[] (elementwise)
		std.foo(int;int *;int *;int *;):::ENTER
		cnt == size(a[]) == size(b[])
		cnt == 100
		a[] elements >= 0
		b[] elements >= 0
		==========
		std.foo(int;int *;int *;int *;):::EXIT1
		cnt == orig(cnt) == size(c[]) == size(a[]) == size(b[])
		a[] == orig(a[])
		b[] == orig(b[])
		cnt == 100
		c[] > a[] (lexically)
		c[] >= a[] (elementwise)
		c[] > b[] (lexically)
		c[] >= b[] (elementwise)

Table 2: Comparison of invariants expected by the programmer and those generated by Daikon.

Note that Daikon does, in fact, create many more invariants than the programmer expected. For example, it reminds us that cnt is equal to the size of all of the arrays passed in, and that the array sizes are not changed during the function. Because the elements in the arrays a[] and b []are randomly generated integers in the range 0 to 9, the programmer might wish to see that 0<=a[] elements <10. However, if we set the option that disallows "trivial invariants," Daikon notes only that the elements of a[] are greater than or equal to 0, regardless of the size of the test suites, because it can not determine if the values are merely data dependencies or if the data is bounded within a certain range by the programmer's intention.

From this example, we infer that Daikon looks for invariants using clues from the values of variables but cannot utilize information from the source code, which it does not understand. Therefore, while Daikon can infer quite a few relationships, it cannot capture some that can easily be found with a review of the code. Also, while c[ ] >= a[ ] (elementwise) and c[ ] >= b[ ] (elementwise) are, in fact, correct invariants and hold through the computation, they are not as strong as they could be and are not closely related to the programmer's intention when executing c[i] = a[i]+b[i] for 0 <= i < 100. We think this limitation can be minimized by taking clues from the source code in addition to trying typical candidates for invariants (such as array size comparisons or inequality relationships between array elements).

5.1.3 Calls to Library Functions

Since standard library calls compose a large portion of the average C program, Daikon should detect invariants in relationships caused by calling library functions in addition to functions defined in the source code. However, we do not believe that, at the moment, Daikon does so. (This is understandable, as it is an extremely difficult problem. Instrumenting the source code that composes the library functions is not a viable option, but it is the only clear solution.) Examining the invariants in **Table 3**, we found that Daikon lacks the capability to derive invariants caused by calls to the standard C libraries (e.g. math.h). Also, it shows the relative weakness of invariants generated when floating-point data is examined.

Code (Bisection and Newton Method) Example that Uses Library Functions	User's Expected Invariants	Invariants Detected by Daikon

int main()
{
.... large iteration ....
newton(x);
 	
}
void newton(double x)
{
int n=0;
double last_f;
double fp,f=function_g(x);
do {
fp=function_gprime(x);
x=x-(f/fp);
last_f=f;
f=function_g(x);
n++;
} while (!((last_f-f)<=FUNC_EPSILON && (last_f-f)>=-FUNC_EPSILON));
return;
}
double function_g(double x)
/* g(x) is x-tan(x) */{
return (x-tan(x));
}
.....
double function_f(double x)
/* f(x) is x^3+2x^2+10x-20 */{
return (((((x+2)*x)+10)*x)-20);
}
.....
}
	for each function: function_g(double x), invariant "return=x-tan(x)," etc.	std.main():::EXIT1
		return == 0
		Exiting
		
		Only one, trivial invariant found.

Table 3: Comparison of invariants expected by the programmer and those generated by Daikon.

5.1.4 Function vs. Statement Implementations

By comparing two implementations of the same program, this example shows that Daikon cannot find interesting relationships between data if the pieces of data involved are neither parameters nor return values for a function. The first version uses a function call to add two values together, while the second version simply adds the two numbers in a single statement – without calling a function. Even though they implement exactly the same function, Daikon cannot detect Hoare-triple like pre-conditions and post-conditions or loop invariants, unless they are revealed using a function. The Daikon manual mentioned this weakness, and we assume that Daikon currently avoids considering local variables because of the difficulty of the problem.

	Function Version	Statement Version

Code	int foo(int a,int b){
	return a+b;
	}
	int main (){
	int k;
	int a,b,c;
	for (k=0;k<10000;k++) {
	a=rand()%100;
	b=rand()%100;
	c=foo(a,b);
	}
	}
		int main ()
		{
		int k;
		int a,b,c;
		for (k=0;k<10000;k++) {
		a=rand()%100;
		b=rand()%100;
		c=a+b;
		}
		}

Invariants Detected	========== std.foo(int;int;):::ENTER a >= 0 b >= 0 ========== std.foo(int;int;):::EXIT1 .... return == a + b ========== std.main():::EXIT2 return == 0 Exiting	========== std.main():::EXIT1 return == 0 Exiting

Table 4: Reveals the lack of invariants captured that are associated with local variables.

5.2 C++

As noted in the Daikon user's manual, the C/C++ front-end (dfec) is fairly robust for C. However, they are still in the design and testing phase for C++. Hence, when using dfec with C++, we were forced to use a subset of the language. To highlight some of our experiences, we discuss two examples that demonstrate the major problems we encountered. The first is an example of the limitations of dfec and the second revisits the problem from 5.1.4 to reveal how an object oriented design methodology helps Daikon.

5.2.1 Compilation Errors

Compiling C++ or C code with g++ after dfec has instrumented it always results in eleven or more warnings, most of which refer to uninitialized values and do not affect operation. However, Daikon has problems handling certain features of the C++ language, including reference return values, reference parameters that involve a class, and overloaded operators. (Most of these errors are related to Daikon's use of "smart pointers," which determine if out-of-scope or non-program memory is being accessed.) For example, in **Table 5**, below, we have an example class that attempts to overload the assignment operator and return an instance of the class by reference. Errors similar to the ones printed below kept us from using "interesting" classes (including lists and trees) that overload operators (including the assignment operator, for a deep copy).

Source Code	Errors Received
class foo { int foo_x; public: foo(); foo &operator=(foo &); }; foo::foo() : foo_x(0) {} foo &foo::operator=(foo &data) { this->foo_x = data.foo_x; return *this; }	daikon-instrumented/ref_ret.cc: In method `foo &foo::operator= (foo &)': daikon-instrumented/ref_ret.cc:5583: cannot convert `data' from type `foo' to type `const void *' daikon-instrumented/ref_ret.cc:5590: cannot convert `data' from type `foo' to type `const void *'  daikon-instrumented/ref_ret.cc:5592: cannot convert `DAIKON_retval_2' from type `foo' to type `const void *

Table 5: Example of C++ syntax that Daikon cannot instrument correctly.

5.2.2 Function vs. Statement Implementations Revisited

In section 5.1.4, we showed that Daikon returns better results on implementations that use functions as often as possible. C++ is more closely related to Java, in that it encourages the use of object oriented programming, and it also encourages the use of lots of object methods. Because of this, Daikon returns more (intermediate) results than either of the examples in 5.1.4, as shown in table [] below. It is completely unnecessary to use a class in the code below, but we wished to solve the same problem as the earlier section(section 5.2.1).

Code	Invariants Returned

class intWrap {
  int x;
 public:
  intWrap();
  void setVal(int newX);
  intWrap operator+(intWrap other);
};
 	
intWrap::intWrap() : x(0) {};
 	
void intWrap::setVal(int newX) {
 x = newX;
}
 	
intWrap intWrap::operator+(intWrap other) {
 intWrap sum;
 sum.x = this->x + other.x;
 return sum;
}
 	
int main () {
 intWrap a, b, c;
 for (int k = 0; k < 10000 ; k++) {
  a.setVal(rand()%100);
  b.setVal(rand()%100);
  c=a+b;
 }
 return 0;
}
	intWrap.intWrap():::ENTER
	this.x == 0
	==========
	intWrap.intWrap():::EXIT1
	this.x == orig(this.x)
	this.x == 0
	==========
	intWrap.operator+(intWrap;):::ENTER
	this.x >= 0
	==========
	intWrap.operator+(intWrap;):::EXIT3
	this.x == orig(this.x)
	this.x >= 0
	==========
	intWrap.setVal(int;):::ENTER
	newX >= 0
	this.x >= 0
	==========
	intWrap.setVal(int;):::EXIT2
	newX == this.x == orig(newX)
	newX >= 0
	orig(this.x) >= 0
	==========
	std.main():::EXIT4
	return == 0

Table 6: Results of running Daikon on a small class related to the examples in 5.1.4.

6 Using Daikon's dfej: Experiments with Java

We performed our experiments on dfej after completing our research into dfec, since our work in architecture required that we use C or C++. Therefore, we had a list of points we wanted to check that had revealed themselves during our earlier observations of Daikon. First of all, since we couldn't explore Daikon's ability to analyze object oriented languages with C++, we wanted to check how it responds to polymorphism and dynamic dispatching. Second, by observing the examples included in the Daikon package, we had noticed that Daikon generates lots of useful invariants for the data structures stack and queue. However, both of them share properties in the sense that the two data structures are strongly related to the size of the array or on boundary conditions, which Daikon is very good at discovering. Therefore, we wished to observe how effective Daikon is on complicated objects like binary search trees or sorted objects. Finally, while working with Daikon, we noticed that both instrumenting programs and running them is very time consuming and a heavy memory load. Since Daikon writes a trace of all variables into a file, we wanted to see if Daikon is robust and can scale to large, resource intensive programs.

Our test programs were selected to test the three areas listed above:

- Polymorphism and Dynamic Dispatching (Section 6.2)
    - Dynamic object maintains only one sub-class type
    - Dynamic object maintains several sub-class types
- Complex ADT (Section 6.1)
    - Queue/Stack: Simple ADT with high correlation to boundary conditions
    - Binary Search Tree/Sorting: ADT with complex semantic invariants and without a strong correlation to boundary conditions
- Large, resource intensive programs (Section 6.3)
    - Large test suites
    - Many methods called
    - Intensive inter-class data interactions

6.1 Simple ADTs vs. ADTs with Complex Semantic Invariants

Two ADT examples implemented using Java are included inside the Daikon tool package. One is a stack implemented using an array, and the other is a queue, also implemented with array. For both of these programs, Daikon generates the invariants in which programmers are most interested: boundary conditions, including the top item of the stack, the size of the queue, etc. However, these invariants are examples of those that Daikon finds easily. However, many ADTs don't emphasize boundary values or the size of the data structure. For example, when speaking of binary search trees, programmers are probably most interested in the semantics correctness of the data and methods. For example, "any node to the left of the current node contains an equal or lesser value than that stored in any node positioned toward the right." Another example is that of ADTs that perform sorting. The developers of such code are not interested in boundary conditions or the size of the array. Instead, they would rather know whether any part of the array is sorted at certain points in the code, for instance.

ADT	Method Name	Interesting Invariants	Invariants Detected by Daikon (Note: Daikon found more invariants than shown in this table.)
Stack	isEmpty( )	return = =true <==> this.topOfStack = = -1 return = = false <==> this.topOfStack > = 0	return = =true <==> this.topOfStack = = -1 return = = false <==> this.topOfStack >= 0
	top( )	return = = this.theArray[this.topOfStack]	return = = this.theArray[this.topOfStack]

	push(Object x)	x = = orig (x) == this.theArray[this.topOfStack] orig(this.topOfStack) = = this.topOfStack – 1	x = = orig (x) == this.theArray[this.topOfStack] orig(this.topOfStack) = = this.topOfStack – 1
	In most cases, the invariants of interest are included in the invariants generated by Daikon.		
Binary Search Tree	findMin( )	return = = the nodes traversed by accessing only left children in the tree	return != null orig (this.root) has only one value. this.root has only one value.
	insert(Object x)	After insertion, x is one of the descendents of this tree.	x = = orig (x) x ! = null this.root = = orig (this.root)
	Although it is, admittedly, difficult to express significant invariants for a BST, Daikon does not generate invariants that reveal any characteristics of this ADT. It detects only trivial invariants.		
Sort	checkSort(Array a)	for any i < size() a[I] <= a[i+1]	a[i] contains no nulls and has only one value.
	SwapReferences(index1, index2)	a[index2]==orig(a[index1]) a[index1]==orgi(a[index2])	a[index2]==orig(a[index1]) a[index1]==orgi(a[index2])
	For the swap function, the invariants detected by Daikon revealed concepts in which the programmer might be interested, but for checkSort, which is a routine that checks whether the array is sorted or not, Daikon detected only trivial, uninteresting invariants.		

Table 7: The comparison of expected invariants and detected invariants for different ADTs.

6.2 Dynamic Dispatching with Invariants vs. Dynamic Dispatching without Invariants

Polymorphism is one of the most important concepts implemented in C++ and Java. Daikon can detect the fixed class type of an object when the object is instantiated as a certain type. However, when the parameter is declared to be an interface or abstract class type, the actual argument can be defined as any type of object that implements the parent class. When the parameter sent is always the same refined subclass, we were uncertain whether or not Daikon would generate an invariant that mentions the type of the argument being passed to the method.

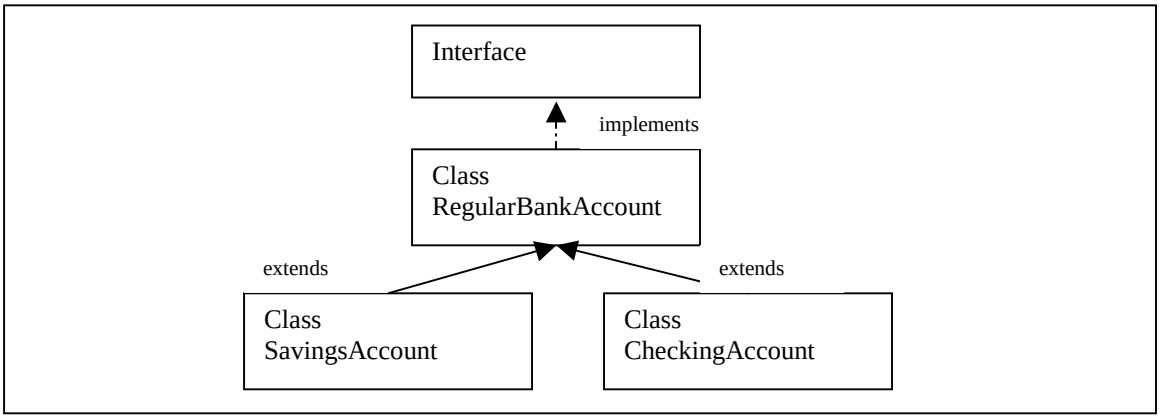

Figure 1: Class relationship diagram for Bank Account.

For example, in the figure above, SavingsAccount and CheckingAccount are both refined subclasses of RegularBankAccount. As descendents of RegularBankAccount, they have methods named getBalance( ) and deposit( ). The type of an instance of these objects is statically declared to be the type of the Interface, BankAccount, but later, it could be defined dynamically to be either SavingsAccount or CheckingAccount.

	Code	Invariants Generated by Daikon for Bank Account "ba"

Subclass type that implements an interface does not change during execution	public void test(BankAccount ba){
	for (int i=0;i<100;i++){
	if (i%3==0) {
	String raowner="RegularAccountOwner";
	ba=new RegularAccount(raowner);
	ba.deposit();
	System.out.println("Balance:"+ba.getBalance());
	}
	}
		Pre-condition
		Ba = = null
		Post-condition
		orig(ba) = = null
		ba has only one value

Subclass type that implements an interface frequently changes during execution.	public void test(BankAccount ba){
	for (int i=0;i<100;i++){
	if (i%3==0) {
	String raowner="RegularAccountOwner";
	ba=new RegularAccount(raowner);
	ba.deposit(i);}
	else if (i%3==1) {
	String chowner="CheckingAccountOwner";
	ba=new CheckingAccount(chowner,120);
	ba.deposit(i); }
	else if (i%3==2) {
	String svowner="SavingAccountOwner";
	ba=new SavingsAccount(svowner,0.2,1500);
	ba.deposit(i); }
	}
	}
		Pre-condition
		Ba = = null
		Post-condition
		orig(ba) = = null
		ba has only one value
		
		(The same as the above case.)

Table 8: Dynamic dispatching example.

In the first example in Table 8, BankAccount "ba" is always instantiated as a RegularAccount, but in the second case, "ba" is instantiated, at different times, as all of the types that implement BankAccount. We carefully examined the post-conditions related to BankAccount "ba" and found that Daikon did not catch the invariant in either case.

6.3 Large-scale Programs

Our target application is a pre-processor for a data mining application that prepares data so that the results can be directly applied to a decision tree algorithm for machine learning. This preprocessor was built for the migration prediction in ubiquitous computing project for the Autumn Quarter 2001 data mining class at the University of Washington.

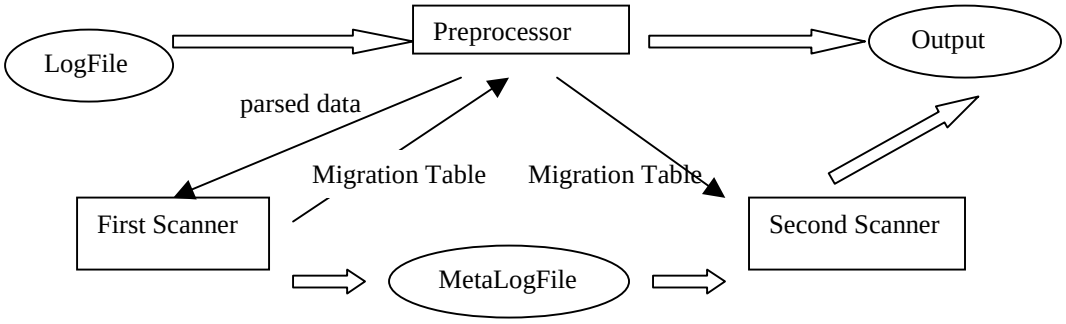

Figure 2: Architecture Diagram for Pre-processor for decision tree algorithm.

This application differs greatly from the toy programs that we used in our experiments. The test suites we used were large and arbitrary. For the pre-processor, the initial data set comes from reading logfiles, which contain a large dataset created from monitoring user interaction with the system. Thus, we believe that the pre-processors' execution will be almost free from data dependencies from the test suites. In addition, the Preprocessor, the First Scanner, and the Second Scanner components interact often, and the parameters the components pass to one another are complex data structures containing parsed data (class Feature Vector, class LocationTime, class MigrationTable, etc). Even inside each component, most computations are done by passing large sets of parameters. This should compensate for Daikon's inability to find invariants involving local variables. Finally, this application is resource and computation intensive, so it will challenge Daikon's ability to scale to large programs.

As a developer of above application, I expected to see interesting invariants that I did not notice during development. However, Daikon detected only invariants that checked boundary conditions and the size of the parameters passed, and none of the detected invariants revealed semantic transformations performed on the parameters. Despite the use of invariant filtering options, Daikon listed more than a thousand invariants that involved NULL and the size of various structures, which were distracting. In addition, the instrumentation and invariant detection stages performed on the first version of the preprocessor source code caused a memory shortage in the Java virtual machine, and I had to modify the source code to reduce the number of iterations performed and to make it less computationally intensive. We believe there should be more improvements to make Daikon applicable to large-scale programs.

7 Flexibility and Configurability

Daikon was designed with the user in mind and can be easily configured using a large number of very useful (and usable) options. They are somewhat difficult to find and use in the command-line version, but the GUI makes Daikon easier to use and reduces the learning curve necessary to get good results. Nevertheless, with or without the GUI, experience with Daikon is important, since there are so many possible configurations. Unfortunately, we found that to get the most useful results, it was necessary to use the "options."

For example, in section 5.1.1, we noted that it was necessary to create and configure the splitter info file that Diakon uses to look for conditional invariants. Another useful option offered is the ability to configure run-time types. Dynamic dispatching can cause Daikon's output to be fairly sparse, since it is difficult to determine the exact type of variables in Java and C++, so dfej parses comments inserted by the programmer. For example, if "/*refined_type: Integer*/ Object element;" is inserted, then dfej can consider more invariants that involve the integer type.

Daikon also supports a series of controls that determine which possible invariants are discarded before output is sent to the user. In some cases, the possibilities that it discards are useful, while it is sometimes helpful to reduce verbosity, to make the remaining output easier to understand. In summary, those control option have following effects:

- Eliminate Redundant Invariants
    - Eliminate an invariant for a specific method if it is directly implied by invariants of the object to which the method belongs.
    - Remove post-condition invariants that are revealed by the pre-condition invariants.
    - Do not display identical invariants applied to different variables.
      If variables are identical, invariants are displayed for only one of them, as some variables are used to store the original values of other variables.
- Filter Invariants by Confidence Level
    - Do not display invariants that contain only constants.
      In many cases, invariants that contain only constants result from dependencies in the test suite, rather than the intention of the code. (As we mentioned in section 5.1.2, this option can remove interesting invariants if it was the programmer's intention to bound the data within a certain range.)
    - Calculate the probability that an invariant holds and eliminate those that describe exceptional or less likely situations.

- Focus on Variables of User Interest
    - Display only invariants that involve variables in which the programmer has registered interest.

Therefore, Daikon's output varies widely depending on the options used. Many of the filters are used by default to reduce the amount of output with which a programmer must deal, which can cause important invariants to be missed entirely. However, this problem is caused by inexperience on the part of the user, rather than by Daikon. The sheer number of configurations that Daikon supports is initially daunting, but it is this flexibility that makes it such a powerful tool, and if used properly, it makes the job of finding invariants much simpler.

8 Possible Improvements

While we used Daikon for our project, we noted both those things that we liked about Daikon and those things that irked us or that we would have liked to have seen. In this section, we air a number of things we think would greatly improve Daikon's usability and appeal, including its ability to handle local variables, scale, and be context-sensitive.

As displayed in Table 4, Daikon operates at a functional granularity. Relationships between local variables are not investigated. Admittedly, such an analysis is, in some sense, already performed, since Daikon tracks the values of parameters and return values, which affect local variables. However, since Daikon also does not handle library functions, many local variables completely escape its notice. In some applications, this can be detrimental, as shown in Table 3. The user manual states that future version of Daikon will include options to perform analysis on local variables, and we would like to encourage them to do so soon.

In addition, the opposite end of the spectrum is another problem. Daikon has difficulty scaling to large problems. As mentioned in section 6, memory became the constraining resource when running a fairly modest data-mining program that had been instrumented with dfej. (Similarly, circular dependencies cause severe problems, as a state-explosion type problem is encountered.) Even in programs that are not especially memory intensive, time can become an issue. After all, instrumenting any program adds four thousand or more lines of code, and function calls become much more expensive. Analyzing local variables, as we suggested above, will only increase the time spent, but a tradeoff between knowledge gained and time consumed certainly exists. The user should be able to affect how much of the code needs to be instrumented or how complete the analysis should be.

Therefore, some context sensitive instrumentation may be needed. Daikon currently supports only two options: considering a specific class type in all contexts or not considering that class at all. If a programmer is interested in a variable of type X in some context, Daikon must instrument all parts of the code that deal with variables of that type. This can cause serious space explosion problems when a programmer tries to instrument all classes in even a small program. These problems, in addition to the severe limitations dfec places on the types of C++ programs that can be instrumented, makes Daikon a much less efficient and attractive system. To make Daikon more widely usable, they will probably have to be addressed.

9 Conclusions

We found Daikon to be a useful tool with a lot of potential, although it is not yet in a state where we believe it can be widely used. Once the technology is improved, many fields could use invariants that Daikon suggests to support verification, testing, and safe software evolution. Daikon fills a niche and was cleverly designed. We particularly liked how its authors have categorized likely invariants and developed the technology to detect those invariants using data traces. In addition, the filtering options they provide hint at the possibility that the user will be able to specify exactly the invariants in which they are interested.

However, some problems exist in the current state of dynamic invariant detection technology. First, Daikon is limited to analyzing a subset of the variables: it only traces parameters and return values. Second, the universe of invariant candidates is extremely limited. We only saw variations upon array size, boundary conditions, and inequality relationships, while we would have liked to see mathematical or semantic relationships, and in general, the granularity of the analysis was too coarse. In addition, Daikon was limited in the size of the applications it could analyze, and finally, when using it or any other dynamic

invariant detection tool, the user must be very careful to differentiate between real invariants and false invariants caused by flawed test suites. For these reasons, among others, while Daikon is an excellent proof of concept, we believe that advances in technology and implementation will be required before dynamic detectors can expect widespread use.

10 References

[1] Michael D. Ernst, "Dynamically Detecting Likely Program Invariants," PhD Dissertation, University of Washington, August 2000.

[2] Michael D. Ernst, http://pag.lcs.mit.edu/daikon/download/doc/daikon_manual_html/daikon.html, February 2002.

[3] Jeremy W. Nimmer and Michael D. Ernst, "Invariant Inference for Static Checking: An Empirical Evaluation," 2001.