The IOA-Daikon Connection: Enabling Dynamic Invariant Discovery in IOA Programs

Gustavo Santos, Toh Ne Win

30 June, 2001

1 Introduction

This document describes the work done during the summer of 2001 in order to connect the IOA Toolkit and Daikon.

1.1 Daikon requirements

The process of detecting invariants with Daikon can be split into two stages: instrumentation/execution and analysis. Every language that can be used with Daikon has to have a "front end" to generate program trace files. This is the instrumentation/execution stage. For example, Java programs are instrumented (modified) by a program called Dfej so that during execution, they generate trace files. The IOA simulator both instruments and executes IOA programs, generating trace files. During the analysis stage, the actual Daikon analyzer examines a set of program trace files and suggests invariants.

The Daikon manual can be consulted for more detail on this topic.

1.2 Improvements summary

The improvements to the IOA Toolkit and in Daikon can be split into two groups:

- Changes to the IOA simulator to extend printing of Daikon-readable program traces and to allow users to specify the amount and type of instrumentation.
- Changes to Daikon to enable analysis of invariants commonly used in IOA and to print invariants in IOA-readible form.

Before this work, the IOA simulator featured a working trace file generator to Daikon for simple numerical state variables. All other variables were represented as strings, so Daikon was not able to test anything other than equality between these variables.

On the Daikon side, the invariants examined were geared towards Java and C programs (e.g. pointer comparisons) and invariant output was not available in IOA format.

2 Changes to the IOA simulator

2.1 Listener interfaces

The first change to the IOA simulator was to modularize its Daikon instrumentation routines. Previously, Daikon instrumentation was done in the same Java class as the one that ran the simulator itself. Besides being unrelated to the main simulator core, Daikon-related code was becoming larger. Hence, all-Daikon related code was moved into a separate Java package, and Antonio Ramirez's Listener design was used as the external interface to the rest of the IOA Toolkit.

This code move allowed later improvements, especially for instrumenting complex IOA data structures, to be done more easily. As a side benefit, the IOA simulator core was pruned and generalized so that many of its features became used in other parts of the IOA toolkit.

2.2 Instrumenting IOA data structures

For Daikon to be a truly useful tool for analyzing IOA programs, it needs to see the contents of all data within a program, especially the insides of data structures. This need is limited by two factors:

- Not all IOA data structures can be easily printed. For example, it is possible to define a graph as an LSL specification and to write Java code to implement it. However, printing the contents of such graphs requires a bit of programming, and there is no easy way to generalize such printing for all data structures - in other words, it may be necessary to write extra Daikon instrumentation code every time a new type of data structure is used.

- Not all printable IOA data structures can be understood by Daikon for their full meaning. Daikon's current design can read integers, strings and arrays of integers and strings [1]. Front ends for Daikon decompose complex data structures into these four basic ones. However, in this decomposition process, connections between parts of larger data structures may be lost.

The simplest improvement to the IOA simulator was to allow Daikon to understand IOA arrays. Since IOA arrays are any mapping from a set to another set, while Daikon arrays are of the traditional variety with numeric indices, we had to restrict the arrays that were instrumented to those readible by Daikon. The following rules were used:

- Any IOA array that does not have a numerical domain (index) is not instrumented as an array.

- An IOA array whose range is numerical is instrumented as an array of numbers.

[1] Note that Daikon can read these four types from its trace files, but it does a more sophisticated internal analysis based on the declared data type of the variable

- The remaining arrays (numerical domain, non numerical range) were instrumented as arrays of strings.

Note that arrays of strings are less useful to Daikon than arrays of numbers, since numerical array contents can be compared better with other numerical variables. A similar rule was used for the IOA types of set and stack.

The weakness of the above method is that it permits only one level of hierarchy: if there is interesting data inside of an array, Daikon can only see it as a single string and has no ability to look inside.

Next, IOA tuples were instrumented for Daikon. Since Daikon does not read in tuples, we used the same convention as that used in other Daikon front ends: create new variables that represent the tuple's fields. So if a tuple called Var has fields a and b, then two new variables are created: Var.a and Var.b. The declared type of each new variable matches that of the original tuple's fields, and when invariants are printed, they are semantically correct. This method also works (unlike the above one for arrays) when a tuple's field is also another complex data structure: if Var.a is another tuple, then the variable Var.a.x could be created.

Finally, for Daikon to see into complex data in arrays and sets, a final instrumentation method was added to specially handle arrays whose range type was a tuple. This was done by also creating new variables. If Var[] is an array of tuples, then new variables that are created will include Var.a[] and Var.b[]. This is a technique adapted from the one used in Dfej.

A major weakness of the above method is that Daikon does not understand that there is a connection between Var.a[] and Var.b[]. To alleviate this problem somewhat, the string representation of Var[] as an array of tuples is also sent to data trace files. A more optimal approach may be to enable Daikon itself to understand the idea of a tuple as a data type.

2.3 Options for the user

The additions described so far to the IOA simulator work without user input (except to specify a flag to the simulator to tell it to generate Daikon trace files). However, not all IOA programs need the same type of instrumentation, and sometimes there are too many variables instrumented, generating too many irrelevant invariants.

We thus implemented an options file so that users can determine what kind of instrumentation they want. The options that can be enabled or disabled are:

- Whether to look into arrays.
- Whether to expand tuples of they are members of arrays
- Ability to ignore a particular variable
- Ability to ignore a particular part of an array-tuple expansion.

3 Changes to Daikon

Toh's note: This is Gustavo's section, so I can't say as much on this.

Changes to Daikon to help with invariants useful in IOA programs:

- Printing of invariants in IOA-readable format, so that the invariants detected by Daikon can be placed back into IOA programs.
- Recognition of sets of data items. Previously, Daikon only knew of arrays as collections, and arrays imply order in the elements. With sets, some kinds of invariant detection used on arrays (such as uniqueness) are disabled, while other types, such as looking at unions and intersections, is implemented.
- Detection of some conditional and biconditional invariants. If invariant A only holds while B holds, then Daikon detects this fact.

4 Bibliography

- Laura Dean, thesis.
- Michael D. Ernst et. al, “Daikon invariant detector”. This is the manual for Daikon.
- Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin, “Dynamically Discovering Likely Program Invariants to Support Program Evolution”. IEEE Transactions in Software Engineering, v27 no. 2, Feb 2001,
- Stephen J. Garland and Nancy A. Lynch. “Using I/O Automata for Developing Distributed Systems”. In Gary T. Leavens and Murali Sitaraman, editors, Foundations of Component-Based Systems, pages 285-312, Cambridge University Press, 2000.
- Michael Tsai, “ADTs for IOA Code Generation”.
- Antonio Ramirez, thesis.