Available online at www.sciencedirect.com

ScienceDirect

Science of Computer Programming 69 (2007) 35–45

Science of Computer Programming

www.elsevier.com/locate/scico

The Daikon system for dynamic detection of likely invariants

Michael D. Ernst*, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, Chen Xiao

MIT Computer Science and Artificial Intelligence Lab, 32 Vassar Street, Cambridge, MA 02139, USA

Received 3 November 2005; received in revised form 31 July 2006; accepted 16 January 2007
Available online 11 October 2007

Abstract

Daikon is an implementation of dynamic detection of likely invariants; that is, the Daikon invariant detector reports likely program invariants. An invariant is a property that holds at a certain point or points in a program; these are often used in assert statements, documentation, and formal specifications. Examples include being constant ($x = a$), non-zero ($x \neq 0$), being in a range ($a \leq x \leq b$), linear relationships ($y = ax + b$), ordering ($x \leq y$), functions from a library ($x = \text{fn}(y)$), containment ($x \in y$), sortedness (x is sorted), and many more. Users can extend Daikon to check for additional invariants.

Dynamic invariant detection runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. Dynamic invariant detection is a machine learning technique that can be applied to arbitrary data. Daikon can detect invariants in C, C + +, Java, and Perl programs, and in record-structured data sources; it is easy to extend Daikon to other applications.

Invariants can be useful in program understanding and a host of other applications. Daikon's output has been used for generating test cases, predicting incompatibilities in component integration, automating theorem proving, repairing inconsistent data structures, and checking the validity of data streams, among other tasks.

Daikon is freely available in source and binary form, along with extensive documentation, at http://pag.csail.mit.edu/daikon/.

Keywords: Daikon; Dynamic analysis; Dynamic invariant detection; Inductive logic programming; Inference; Invariant; Likely invariant; Program understanding; Specification; Specification mining

1. Introduction

This paper presents Daikon — a full featured and robust implementation of dynamic invariant detection. Invariants explicate data structures and algorithms and are helpful for manual and automated programming tasks, from design to maintenance. For example, they identify program properties that must be preserved when modifying code. Despite their advantages, invariants are usually missing from programs. An alternative to expecting programmers to fully annotate code with invariants is to automatically infer likely invariants from program executions. For certain important tasks, dynamically-inferred properties are preferable to a human-written specification.

[*] Corresponding author.
E-mail addresses: mernst@csail.mit.edu (M.D. Ernst), jhp@csail.mit.edu (J.H. Perkins).
URL: http://pag.csail.mit.edu/daikon/ (Daikon invariant detector distribution).

0167-6423/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2007.01.015

Daikon's output is a set of likely invariants that are statistically justified by an execution trace. The result can be used as documentation, added to the program source as assertions, or used as input to other tools (see Section 4).

Daikon is freely available for download from http://pag.csail.mit.edu/daikon/. The distribution includes both source code and documentation, and Daikon's license permits unrestricted use. This paper introduces Daikon, but its user and developer manuals should be consulted to obtain the most complete information.

2. Example

As an example, consider a Java class `StackAr` that implements a stack with a fixed maximum size. It has the following fields:

```
Object[] theArray;   // Array that contains the stack elements.
int      topOfStack; // Index of top element. -1 if stack is empty.
```

and implements the following methods:

```
void push(Object x)   // Insert x
void pop()            // Remove most recently inserted item
Object top()          // Return most recently inserted item
Object topAndPop()    // Remove and return most recently inserted item
boolean isEmpty()     // Return true if empty; else false
boolean isFull()      // Return true if full; else false
void makeEmpty()      // Remove all items
```

`StackAr` can be exercised via the simple test class `StackArTester`. For example:

```
java DataStructures.StackArTester
```

The following command instruments the same run of `StackAr` and pipes the resulting execution trace to Daikon.

```
java daikon.Chicory --daikon DataStructures.StackArTester
```

(Chicory is Daikon's instrumenter for Java.) The likely invariants are written to standard output (i.e., the console) and also saved in a serialized file for later processing.

C/C + + programs are handled in a similar fashion. If a C/C + + program is normally run with the command

```
./program -f input
```

then the command

```
kvasir-dtrace --dtrace-file=- ./program -f input | java daikon.Daikon
```

will instrument the program using Kvasir (an instrumenter for C/C + +) and pipe the resulting execution trace to Daikon.

Daikon's output consists of procedure pre- and post-conditions and also object invariants which hold at every public method entry and exit. Fig. 1 shows a portion of Daikon's output, including the object invariants and the procedure invariants for the constructor and the method `isFull`. The output is described below.

```
this.theArray != null
```

The reference `theArray` was never observed to be null after it was set in the constructor. Methods can thus safely reference it without checking for null.

```
this.theArray.getClass() == java.lang.Object[].class
```

The runtime class of `theArray` is `Object[]`. Java permits the runtime type to be a subclass of the declared type, but that never happened in `StackAr`.

```
this.topOfStack >= -1
this.topOfStack <= this.theArray.length - 1
```

`topOfStack` is between -1 and the maximum array index, inclusive. Thus, it indicates the last filled array element, not the next one to be filled.

Object invariants for StackAr

```
this.theArray != null
this.theArray.getClass() == java.lang.Object[].class
this.topOfStack >= -1
this.topOfStack <= this.theArray.length - 1
this.theArray[0..this.topOfStack] elements != null
this.theArray[this.topOfStack+1..] elements == null
```

Pre-conditions for the StackAr constructor

```
capacity >= 0
```

Post-conditions for the StackAr constructor

```
orig(capacity) == this.theArray.length
this.topOfStack == -1
this.theArray[] elements == null
```

Post-conditions for the isFull method

```
this.theArray == orig(this.theArray)
this.theArray[] == orig(this.theArray[])
this.topOfStack == orig(this.topOfStack)
(return == false) <==> (this.topOfStack < this.theArray.length - 1)
(return == true)  <==> (this.topOfStack == this.theArray.length - 1)
```

Fig. 1. Daikon output for the `StackAr` program. The figure shows likely object invariants and the pre- and post-conditions for the constructor and `isFull` method. There are no pre-conditions for `isFull`.

`this.theArray[0..this.topOfStack] elements != null`

All of the stack elements are non-null. This turns out to be a result of the test suite (which never pushes null on the stack) and not a requirement of `StackAr` itself. It is still valuable to draw a programmer's attention to this fact, as it points out a deficiency in the test suite.

`this.theArray[this.topOfStack+1..] elements == null`

All of the elements in the array that are not currently in the stack are null. We can infer that the pop method nulls out the entry after returning it, which is good practice as it permits those elements to be garbage collected.

Likely invariants are also found at each method entry and exit. These correspond to the pre- and post-conditions for the method. There is one pre-condition for the `StackAr` constructor:

`capacity >= 0`

`StackAr` was never created with a negative capacity. (This particular constructor implementation would fail if it is supplied a negative capacity.)

The post-conditions for the `StackAr` constructor are:

`orig(capacity) == this.theArray.length`

The size of the array that will contain the stack is equal to the specified capacity.

`this.topOfStack == -1`

`this.theArray[] elements == null`

Initially the stack is empty and all of its elements are null.

There are no pre-conditions for the `isFull` method other than the object invariants. The post-conditions are:

```
this.topOfStack == orig(this.topOfStack)
this.theArray == orig(this.theArray)
this.theArray[] == orig(this.theArray[])
```

Neither the `topOfStack` index, nor the reference to the `theArray` array, nor the contents of the array are modified by the method. In other words, the method is pure (an observer method).

```
(return == true) <==> (this.topOfStack == this.theArray.length - 1)
```

When `isFull` returns true, `topOfStack` indexes the last element of `theArray`.

```
(return == false) <==> (this.topOfStack < this.theArray.length - 1)
```

When `isFull` returns false, `topOfStack` indexes an element prior to the last element of `theArray`.

3. Key features of Daikon

The following subsections detail Daikon's key features, which allow it to be used in a wide variety of contexts.

3.1. Input from many programming languages and other data formats

Daikon can compute likely invariants for programs written in C, C + +, Java, and Perl (see Section 5.1). Daikon can also process data from spreadsheets such as Excel using CSV (comma separated value) format; this is convenient when running Daikon on data that was not generated by a program. It is relatively easy to support new programming languages and data formats, as several users have done (see Section 5.3).

3.2. Rich output

Daikon's primary goal is to report expressive and useful output. Which invariants are reported depends on the grammar of invariants that are expressible by the invariant detector, the variables over which the invariants are checked, and the program points at which the invariants are checked. We discuss each of these factors below.

Grammar of properties. Daikon checks for 75 different invariants including those mentioned in the abstract, and users can easily extend this list (see Section 5.3). Checking a large number of invariants makes it more likely that the output will contain the facts that are needed by a human or a tool; however, it also increases the runtime of the invariant detector. Daikon includes optimizations that allow it to support a large number of possible invariants (see Section 5.2).

Daikon also can report conditional invariants or implications, such as $this.left \neq null \Rightarrow this.left.value \leq this.value$. Daikon has certain built-in predicates that it uses for finding implications; examples are which return statement was executed in a procedure and whether a boolean procedure returns true or false. Additionally, Daikon can read predicates from a file and find implications based on those predicates [1]. The predicates can be produced manually or automatically using tools distributed with Daikon, such as by static analysis of the program or by cluster analysis of the values in the execution trace.

Grammar of variables. The invariants that an invariant detector can express must be instantiated over particular variables or other values. For example, $x \in y$ is instantiated over two variables, of which the second must be a collection. Daikon can produce invariants over:

- procedure parameters and return values
- pre-state values (expressed as `orig` in Fig. 1), which permits reports of procedure side effects and input–output relationships
- global variables
- fields (given an object `a`, useful additional information appears in `a.f`, `a.f.g`, etc.)
- results of calls to side-effect-free (pure, observer) methods, such as `size()`
- additional expressions called derived variables (25 in all). For example, given array `a` and integer `lasti`, then invariants over `a[lasti]` may be of interest, even though it is not a variable and may not even appear in the program.

These variable types can be combined; for example, `a.b[myObject.getX()]`. A Daikon switch controls whether only public fields/methods are considered (giving an external, client view of the data structure), or private fields/methods are included (giving an internal, implementation view of the data structure). Another switch controls the depth for examining fields and pure method calls.

Program points. Daikon checks for invariants at procedure entries and exits, resulting in likely invariants that correspond to pre- and post-conditions. It computes likely invariants at each procedure exit (i.e., `return` statement) and also at an *aggregate* exit point (as viewed by a client) by generalizing over the individual exit points. Likely object or class invariants are also computed at an aggregate program point (*object point*). The object point generalizes over all objects that are observed at entry to and exit from public methods of a class, that are passed into or returned from methods of other classes, or that are stored in object fields.

3.3. Scalable

Despite the rich output described in Section 3.2, Daikon is scalable to non-trivial programs. For example, Daikon was used to generate partial consistency specifications (see Section 4.6) for the Center-TRACON Automation System (CTAS) developed at the NASA Ames research center [2]. CTAS is a set of air-traffic control tools that contains over 1 million lines of C and $C++$ code. Applying Daikon to large programs often requires focusing Daikon on a subset of the program, or reducing the number of derived variables or even invariants that Daikon considers. All of these can be done via command-line arguments.

3.4. Invariant filtering

Dynamic invariant detection tests potential invariants against observed runtime values. As with any dynamic analysis or machine learning technique, there is a possibility of overfitting (overgeneralizing) by reporting properties that are true of the training runs but are not true in general. For example, if there are an inadequate number of observations of a particular variable, patterns observed over it may be mere coincidence. Daikon mitigates such problems by computing a null hypothesis test (the probability that a property would appear by chance in a random input), then reporting the property only if its probability is smaller than a user-defined confidence parameter.

Redundant invariants. Any invariant that is logically implied by other invariants can be removed from the output with no loss of information. For example, consider the object invariant `this.topOfStack <= this.theArray.length-1` from Fig. 1. The invariant `this.topOfStack < this.theArray.length` is also true but is not reported. As much as possible, Daikon does not even compute redundant invariants. This removes them both from the output and improves Daikon's performance (see Section 5.2). Some redundancies that cannot be handled by the optimizations are removed from the output by post-processing.

Abstract types. Many variables in a program are not related to one another. For example, variable `temp` may contain the current temperature while `time` contains the number of seconds since 1970. Even though invariants may exist over these variables (e.g., `temp < time`), those invariants are not interesting. Daikon can use abstract type information [3] to restrict invariants to related variables. The Daikon distribution includes tools that compute abstract types.

3.5. Output formats

Daikon supports a wide variety of output formats. There are also easy-to-add additional formats (see Section 5.3).

Daikon. Default output format, with quantifiers and other features.
DBC. Design by contract format for Parasoft's Jtest tool [4].
ESC. Extended static checker format for use by the ESC/Java tool [5].
IOA. IOA language for modeling input/output automata, which can also be converted into the language of the Isabelle theorem prover.
Java. Java expressions usable in a Java program, for instance as assertions.
JML. Java Modeling Language [6] format, used by many tools [7].
Repair. Format used by automated data structure repair tools [2].
Simplify. Format of the Simplify automated theorem prover [8].

The `annotate` tool included with Daikon inserts likely invariants into source code as annotations.

3.6. *Portable*

Daikon is portable to a wide variety of platforms. The Daikon inference engine and the Java instrumenter are written in Java, and the Perl instrumenter is written in Perl, languages which are highly portable. One C instrumenter is specific to Linux/x86, while a second is portable to any platform for which Purify is available, including Windows, Linux, and several commercial Unix variants.

4. Uses

The likely invariants produced by Daikon are a dynamically-generated analogue of a program specification. Specifications are widely acknowledged as being valuable in many aspects of program development, including design, coding, verification, testing, optimization, and maintenance. They also enhance programmers' understanding of data structures, algorithms, and program operation. In practice, however, formal specifications are rarely available, because they are tedious and difficult to write, and when available they may fail to capture all of the properties on which program correctness depends.

A dynamically-generated specification may be different from an ideal static specification. The dynamic specification will reveal information not only about the goal behavior, but also about the particular implementation and about the environment (inputs) under which the program was run. For many tasks (such as improving a test suite and certain debugging tasks), the dynamic information can be even more helpful than the static.

4.1. *Documentation*

Invariants characterize certain aspects of program execution and provide documentation of a program's operation, algorithms, and data structures. As such, they assist program understanding, some level of which is a prerequisite to every programming-related task. Information that is automatically extracted from the implementation is guaranteed to be up to date.

Automatically-inferred invariants are useful even for code that is already documented with comments, assert statements, or specifications. The inferred invariants can check or refine programmer-supplied invariants; program self-checks are often out of date, ineffective, or incorrect [9].

4.2. *Avoiding bugs*

Invariants can protect a programmer from making changes that inadvertently violate assumptions upon which the program's correct behavior depends. If the original invariant is not documented, much less the dependence, then it is all too easy for a programmer to violate it, introducing a bug in a distant part of the program. Program maintenance introduces errors [10,11] and many of these are due to violating invariants. When presented with Daikon's output, programmers are able to avoid breaking such invariants [12].

4.3. *Debugging*

Although it is better to avoid errors in the first place, invariants can also assist in debugging. Raz used Daikon to check for consistent inputs in data streams [13]. Many authors have used inconsistencies in inferred invariants between training runs and actual runs as an indicator of program bugs that should be brought to the attention of humans [14–16].

Dynamically detected invariants can also help in understanding or localizing errors. Groce uses differences between succeeding and failing runs to give a concise explanation of test failures [17]. Xie compares behaviors of internal components to localize differences between program versions and isolate errors [18]. Liblit gives a light-weight methodology for determining which properties are indicative of an error [19]. Brun uses machine learning over correct and faulty programs to predict what properties in a different program are most likely to indicate latent errors [20].

4.4. *Testing*

Dynamically detected invariants can reveal as much about a test suite as about the program itself, because the properties reflect the program's execution over the test suite. This information is useful in testing. Insufficient coverage of program values is a deficiency in the test suite, even if the test suite covers every statement or path of the program. For example, "$x > 0$" can indicate that zero and negative values were never tried, and "$x < 10$" that large values were never tried. Furthermore, these properties indicate exactly what inputs are needed to improve the test suite. The inferred invariants can also be used as a type of coverage metric such as for test selection and prioritization; more coverage yields a better test suite [21,22]. Dynamically detected invariants can also assist in throwing out meaningless tests (that violate pre-conditions) and inferring when a test has failed (by violating post-conditions) [23].

4.5. Verification

Dynamic invariant detection proposes likely invariants based on program executions, but the resulting properties are not guaranteed to be true over all possible executions. Static verification checks that properties are always true, but it can be difficult and tedious to select a goal and to annotate programs for input to a static checker. Combining these techniques overcomes the weaknesses of each: dynamically detected invariants can annotate a program or provide goals for static verification, and static verification can confirm properties proposed by a dynamic tool.

We have experimented with several program verifiers. With Java programs and ESC/Java [5], Daikon's precision (percentage of properties provable) was over 95%, of which the unprovable ones were generally due to prover incompleteness, and its recall (percentage of properties needed for program verification found) was over 90% [24]. Furthermore, users who were given the Daikon output did statistically significantly better on a program verification task [25]. With proof assistant tools, about 90% of lemmas and tactics were generated automatically [26], for proofs of three distributed algorithms.

4.6. Data structure and control structure repair

Corrupt data structures cause incorrect or unpredictable program execution. Data structure repair updates corrupt data structures, enabling the program to continue to execute acceptably. However, it requires user-provided specifications. Daikon's output has permitted the entire process to be automated while permitting human review. The result was more robust versions of programs, at reduced cost compared to manual generation [2].

A related use is in programs that dynamically choose modalities in response to environmental inputs. Dynamically detected invariants from training runs can indicate which modalities are correlated with which external conditions. At runtime, occasionally overriding the program's (inconsistent) actions improved the performance of programs in a real-world competition by over 50% [27].

5. Architecture

Daikon discovers likely invariants from program executions by instrumenting the target program to trace certain variables, running the instrumented program, and inferring invariants over both the instrumented variables and derived variables not manifesting in the program. All of the steps are fully automatic (except for requirements of running the target program, such as its inputs), and in fact the user issues only a single command that is much like running the original program (see Section 2).

5.1. Instrumenters

Instrumenters are language specific. Currently Daikon has instrumenters for $C/C++$, Java, and Perl programs. Some of the instrumenters are source based: they create a new version of the source that is then compiled, linked, and run. Other, binary-based instrumenters operate directly upon the executable: they are easier to use but are more likely (at least in the case of compiled languages) to be platform specific.

The basic task of the instrumenter is to add instructions to the target program so that it will output the values of variables (Section 3.2). The traces are piped directly to the inference engine, avoiding the necessity of disk storage. Alternately, the traces can be output to a file, permitting repeated processing or combined processing of multiple runs. In order to obtain good coverage, a target program sometimes needs to be run multiple times over separate inputs. In languages such as $C/C++$, it is not possible to determine if a pointer or array index is valid simply by examining it. In this case, the instrumenter must augment the target program to determine whether each pointer refers to valid memory and the extent of each array.

5.2. *Inference engine*

The inference engine reads the trace data produced by an instrumenter and produces likely invariants. Separating the language-specific instrumenters from the inference engine allows it to be portable and also makes it easier to extend Daikon to new sources of data. Note that while *Daikon* is the name of the entire system, it is also used to refer specifically to the inference engine.

The essential idea is to use a generate-and-check algorithm to test a set of potential invariants against the traces. Daikon reports those invariants that are tested to a sufficient degree without falsification. As with other dynamic approaches such as profiling, the accuracy of the results depends in part on the quality and completeness of the test cases. Even modest test suites produce good results in practice [25,24], and techniques exist for creating good test suites for invariant detection [21,28,22].

A simple algorithm for dynamic invariant detection represents each invariant in the invariant grammar explicitly: it initially assumes that all potential invariants are true and tests each one against each sample. Any invariant contradicted by a sample is discarded. Any invariants remaining at the end of processing are reported. Unfortunately, this algorithm does not scale well.

Daikon is enhanced with a number of optimizations that allow it to scale to both large numbers of invariants and programs of non-trivial size. These optimizations are based on the fact that the simple algorithm checks and reports more invariants than necessary. Many invariants are redundant and do not need to be checked. Together the optimizations reduce the number of invariants that need to be checked by 99% [29].

There are four major optimizations.

- **Equal variables.** If two or more variables are always equal, then any invariant that is true for one of the variables is true for each of the variables. For example, if $x = y$, then for any invariant f, $f(x)$ implies $f(y)$.
- **Dynamically constant variables.** A dynamically constant variable has the same value at each observed sample. The invariant $x = a$ (for constant a) makes any other invariant over (only) x redundant. For example, $x = 5$ implies $odd(x)$ and $x \geq 5$. Likewise for combinations of variables: $x = 5$ and $y = 6$ imply both $x < y$ and $x = y - 1$.
- **Variable Hierarchy.** Some variable values contribute to invariants at multiple program points. For example, values observed at (public) method exits affect not only method post-conditions but also object invariants. For two program points A and B, if all samples for B also appear at A, then any invariant true at A is necessarily true at B and is redundant at B.
- **Suppression of weaker invariants.** An invariant is *suppressed* if it is logically implied by some set of other invariants. (The previous three examples of redundancy are special cases of this one.) For example, $x > y$ implies $x \geq y$, and $0 < x < y$ and $z = 0$ imply $x \text{ div } y = z$.

5.3. *Extensibility*

A primary design goal for Daikon is extensibility, since dynamic invariant detection is a new and active field. Daikon is directly extensible in these ways:

- **Instrumenters.** An instrumenter for a new language or for other data merely needs to output files in Daikon's input format. For example, instrumenters have been added for spreadsheet data, calls to Linux library routines [30], the Java PathFinder model checker [17], and online data feeds [13].
- **Invariants.** Each property in Daikon's invariant grammar is implemented as a subclass of the abstract base class `Invariant`, with further subclasses grouping similar invariants. These classes provide most of the necessary code for an invariant. A new invariant must provide basic functionality such as the `check_modified` method, which checks whether a tuple of values is consistent with the invariant.
- **Derived variables.** A derived variable is an expression that Daikon treats as a variable (see Section 3.2). As with invariants, the class structure makes it easy to add new derived variables by defining a few methods.
- **Output formats.** New output formats (see Section 3.5) require only adding a new format method to each invariant.
- **Suppressions.** As noted in Section 5.2, an invariant is suppressed if it is logically implied by some set of other invariants. Each type of invariant can specify a set of possible *suppressions*. Each suppression is a set of antecedent invariants that together imply the invariant. For example, $\{\{x = y, z = 1\}, \{x = z, y = 1\}\}$ is a set of two suppressions for the invariant $x = y \cdot z$. An invariant is suppressed if all of the antecedents in any suppression hold. A new suppression is added by listing the antecedent invariants that make up the suppression.

- **Modular.** Daikon is designed to make it easy to use as part of another tool. Its entry points all have a corresponding `mainHelper` method that can be called programmatically and that never calls `System.exit()`. Likely invariants are written to a serialized file, for convenient programmatic use. More than 100 configuration options (documented in the user manual) control Daikon's behavior. Daikon is open source, permitting users to make changes (and to contribute them to the project).

6. Development process

Daikon is intended as a long-term, full-featured, robust research tool. Its development process is designed to ensure that this is possible despite the challenges of the academic research environment (e.g., many short-term developers, each with a very specific focus). The keys to the process are:

- **Monthly releases.** A new version of Daikon is released every month. This ensures that the build process is clean and that the documentation is up to date. It also serves as an incentive to address problems in a timely fashion.
- **Documentation.** Daikon has extensive, and frequently updated, user (156 pages) and developer (129 pages) manuals in addition to the technical papers that describe its algorithms and uses. Portions of the manual are automatically created from source code documentation to ensure accuracy.
- **Support.** Several moderated mailing lists exist for users to obtain help. Most problems are attended to within a few days. If a change is required, that change is normally made available immediately and is also included in the next release.
- **Regression tests.** Significant regression tests for both the inference engine and the instrumenters are run nightly, and by the developers before committing code changes. Daikon has been used as a test subject in over half a dozen published papers because of its CVS history and its regression tests.

7. Comparison to related tools

This section describes the most closely related other implementations of dynamic invariant detection. We regret that space constraints prevent discussion of every related tool. While the other implementations use the simple algorithm described in Section 5.2 without Daikon's optimizations, some of them run faster than Daikon, primarily because Daikon checks many more invariants (for example, millions of times more, on flex [29]).

The Agitator product developed by Agitar was inspired by Daikon [31]. Agitar performs dynamic invariant detection in order to inform users about tests, and to improve those tests. The results are called "observations", they include equality ($x = y$), range ($-10 \leq x \leq 10$), non-null ($x \neq \mathit{null}$), equality (.equals), and properties gleaned from the user's source code. Agitar won the Wall Street Journal's 2005 Software Technology Innovation Award.

The DIDUCE tool [14] for Java programs checks one unary invariant, which indicates all previously seen values of every bit of the value. At each program point (a field or array reference, or a procedure call), that invariant is checked for three values: a variable's current and previous values and their difference. As an invariant is weakened (new values are observed in a given bit), a message is printed. A user can look for weakenings that are printed on erroneous runs or just before an error occurs, to find rare corner cases. The tool was used to help explain several known errors and to reveal two new errors.

The IODINE tool [32] extracts likely design properties from hardware designs using dynamic analysis. It looks for such properties as state machine protocols, request–acknowledge pairs, and mutual exclusion between signals. IODINE was used to find dynamic invariants for one block of a dual-core SPARC$^{\text{TM}}$ processor design.

Remote program sampling [19] is a light-weight mechanism that evaluates two properties (one unary and one binary, but instantiated for a linear rather than quadratic number of variable pairs) at assignments in C programs (plus the predicate at each branch), counting the number of times that each property is satisfied. The properties are checked probabilistically: on most executions of a program point, property checking is skipped. Any of the other implementations could be so extended, sacrificing soundness (over the test suite) but gaining performance. The results are processed statistically to indicate which properties are best correlated with faults and thus most likely to be indicative of faults. The tool can provide early warning of errors, if one of the relatively simple properties indicates a bug. In an experiment, the tool rediscovered 7 known errors and found one new one.

Henkel and Diwan [33,34] have built a tool that discovers algebraic specifications, which relate the meaning of sequences of code operations, such as "pop(push(x,stack))=stack". The tool generates many terms (test cases) from the signature of a Java class and proposes equations based on the results of the tests. The tool also proposes and tests generalizations.

The SPIN model checker has been extended to check whether two variables are related by $=$, $<$, $>$, $\leq$, or $\geq$ [35]. The output is a graph with variables at the nodes and edges labeled by the comparison relations.

Several researchers have inferred, from program or system traces, finite state automata that represent the permitted transitions [36–39]. Specifications written in the form of automata are complementary to the formula-based program properties that are generated by a dynamic invariant detector.

Perracotta [40] is similar except that it looks for a set of extensions to the response property pattern rather than an automaton.

8. Conclusion

Daikon is a full-featured, scalable, robust tool for dynamic invariant detection. It supports multiple programming languages and is readily extensible to other languages and sources of data. The likely invariants that it produces have been used in contexts ranging from refactoring to testing to data structure repair, and many more.

Daikon is freely available for download from http://pag.csail.mit.edu/daikon/. The distribution includes both source code and extensive documentation, and Daikon's license permits unrestricted use.

Supplementary data

Supplementary data associated with this article can be found, in the online version, at doi:10.1016/j.scico.2007.01.015.

References

[1] N. Dodoo, L. Lin, M.D. Ernst, Selecting, refining, and evaluating predicates for program analysis, Tech. Rep. MIT-LCS-TR-914, MIT Lab for Computer Science, 21 July, 2003.

[2] B. Demsky, M.D. Ernst, P.J. Guo, S. McCamant, J.H. Perkins, M. Rinard, Inference and enforcement of data structure consistency specifications, ISSTA (2006) 233–243.

[3] P.J. Guo, J.H. Perkins, S. McCamant, M.D. Ernst, Dynamic inference of abstract types, ISSTA (2006) 255–265.

[4] Parasoft Corporation, Jtest version 4.5. http://www.parasoft.com/.

[5] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, R. Stata, Extended static checking for Java, PLDI (2002) 234–245.

[6] G.T. Leavens, A.L. Baker, C. Ruby, JML: A notation for detailed design, in: Behavioral Specifications of Businesses and Systems, Kluwer Academic Publishers, Boston, 1999, pp. 175–188.

[7] L. Burdy, Y. Cheon, D. Cok, M.D. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, E. Poll, An overview of JML tools and applications, STTT 7 (3) (2005) 212–232.

[8] D. Detlefs, G. Nelson, J.B. Saxe, Simplify: A theorem prover for program checking, Tech. Rep. HPL-2003-148, HP Labs, Palo Alto, CA, 23 Jul., 2003.

[9] N.G. Leveson, S.S. Cha, J.C. Knight, T.J. Shimeall, The use of self checks and voting in software error detection: An empirical study, IEEE TSE 16 (4) (1990) 432–443.

[10] M. Ohba, X.-M. Chou, Does imperfect debugging affect software reliability growth? ICSE (1989) 237–244.

[11] T.L. Graves, A.F. Karr, J.S. Marron, H. Siy, Predicting fault incidence using software change history, IEEE TSE 26 (7) (2000) 653–661.

[12] M.D. Ernst, J. Cockrell, W.G. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, IEEE TSE 27 (2) (2001) 99–123.

[13] O. Raz, P. Koopman, M. Shaw, Semantic anomaly detection in online data sources, ICSE (2002) 302–312.

[14] S. Hangal, M.S. Lam, Tracking down software bugs using automatic anomaly detection, ICSE (2002) 291–301.

[15] B. Pytlik, M. Renieris, S. Krishnamurthi, S.P. Reiss, Automated fault localization using potential invariants, AADEBUG (2003) 273–276.

[16] L. Mariani, M. Pezzè, A technique for verifying component-based software, TACoS (2004) 17–30.

[17] A. Groce, W. Visser, What went wrong: Explaining counterexamples, SPIN (2003) 121–135.

[18] T. Xie, D. Notkin, Checking inside the black box: Regression testing based on value spectra differences, ICSM (2004) 28–37.

[19] B. Liblit, A. Aiken, A.X. Zheng, M.I. Jordan, Bug isolation via remote program sampling, PLDI (2003) 141–154.

[20] Y. Brun, M.D. Ernst, Finding latent code errors via machine learning over program executions, ICSE (2004) 480–490.

[21] M. Harder, J. Mellen, M.D. Ernst, Improving test suites via operational abstraction, ICSE (2003) 60–71.

[22] T. Xie, D. Notkin, Tool-assisted unit test selection based on operational violations, ASE (2003) 40–48.

[23] C. Pacheco, M.D. Ernst, Eclat: Automatic generation and classification of test inputs, ECOOP (2005) 504–527.

[24] J.W. Nimmer, M.D. Ernst, Automatic generation of program specifications, ISSTA (2002) 232–242.

[25] J.W. Nimmer, M.D. Ernst, Invariant inference for static checking: An empirical evaluation, FSE (2002) 11–20.

[26] T. Ne Win, M.D. Ernst, S.J. Garland, D. Kırlı, N. Lynch, Using simulated execution in verifying distributed algorithms, VMCAI (2003) 283–297, New York, NY.

[27] L. Lin, M.D. Ernst, Improving adaptability via program steering, ISSTA (2004) 206–216.

[28] N. Gupta, Z.V. Heidepriem, A new structural coverage criterion for dynamic detection of program invariants, ASE (2003) 49–58.

[29] J.H. Perkins, M.D. Ernst, Efficient incremental algorithms for dynamic detection of likely invariants, FSE (2004) 23–32.

[30] S. McCamant, M.D. Ernst, Early identification of incompatibilities in multi-component upgrades, ECOOP (2004) 440–464.

[31] M. Boshernitsan, R. Doong, A. Savoia, From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing, ISSTA (2006) 169–179.

[32] S. Hangal, N. Chandra, S. Narayanan, S. Chakravorty, Iodine: A tool to automatically infer dynamic invariants for hardware designs, DAC (2005) 775–778.

[33] J. Henkel, A. Diwan, Discovering algebraic specifications from Java classes, ECOOP (2003) 431–456.

[34] J. Henkel, A. Diwan, A tool for writing and debugging algebraic specifications, ICSE (2004) 449–458.

[35] M. Vaziri, G. Holzmann, Automatic detection of invariants in Spin, SPIN (1998) 129–138.

[36] J.E. Cook, A.L. Wolf, Event-based detection of concurrency, FSE (1998) 35–45.

[37] J.E. Cook, A.L. Wolf, Discovering models of software processes from event-based data, ACM TOSEM 7 (3) (1998) 215–249.

[38] G. Ammons, R. Bodík, J.R. Larus, Mining specifications, POPL (2002) 4–16.

[39] J. Whaley, M. Martin, M. Lam, Automatic extraction of object-oriented component interfaces, ISSTA (2002) 218–228.

[40] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, M. Das, Perracotta: Mining temporal API rules from imperfect traces, ICSE (2006) 282–291.