Automatic Dimension Inference and Checking for Object-Oriented Programs

Sudheendra Hangal and Monica S. Lam
Computer Science Department
Stanford University, CA 94305
{hangal,lam}@cs.stanford.edu

This paper introduces UniFi, a tool that attempts to automatically detect dimension errors in Java programs. UniFi infers dimensional relationships across primitive type and string variables in a program, using an inter-procedural, context-sensitive analysis. It then monitors these dimensional relationships as the program evolves, flagging inconsistencies that may be errors. UniFi requires no programmer annotations, and supports arbitrary program-specific dimensions, thus providing fine-grained dimensional consistency checking. UniFi exploits features of object-oriented languages, but can be used for other languages as well. We have run UniFi on real-life Java code and found that it is useful in exposing dimension errors. We present a case study of using UniFi on nightly builds of a 19,000 line code base as it evolved over 10 months.

1. Introduction

Dimensional analysis is a simple and well-understood way of checking physics equations for consistency. However, programming languages have poor support for checking dimensional consistency within programs. The work reported in this paper is motivated by the many software errors we have created, debugged, or otherwise encountered that could have been caught by a simple dimension or unit check.

1.1 Dimensions in Programs

While dimensions in physics are traditionally associated with physical quantities like mass, length and time, program variables have dimensions in a broader sense. They represent sizes, dates, colors, IDs, counts, positions, masks, ports, flags, states, file names, host names, addresses, properties, messages, and so on. Consider the following constructor declaration in the Java library class `java.awt.MouseWheelEvent`:

```
MouseWheelEvent(Component source,
    int id, long when, int modifiers,
    int x, int y, int clickCount,
    boolean popupTrigger,
    int scrollType, int scrollAmount,
    int wheelRotation);
```

The interface of this method alone involves several variables with independent dimensions, such as id, modifier, time, coordinate, number of clicks and scroll type. While even moderately sized programs have hundreds of dimensional associations, there is no protection provided to the programmer to keep unrelated dimensions from interacting, or to ensure that they are combined only in dimensionally sound ways.

Dimensions are currently coarsely simulated with programming language types, which do not provide adequate granularity. For instance, an integer variable representing a network port can be interchanged with another integer representing a graphics color, and a string variable holding a file name may be used in place of a host name; in neither case will a conventional type checker complain. To get the benefit of dimensional analysis, the programmer would have to go through a cumbersome process of defining custom types for each dimension associated with a program, and to specify legal ways in which they may interact. This process is especially difficult and unnatural for values of primitive types (e.g. integers and floating point numbers) and strings, which is why we focus exclusively on these types in this work.

Programmers often do have an implicit understanding about dimensions in the program, as indicated by the fact that variable names often refer to dimensions. However, without automatic dimension checking, it is easy for programmers to create obvious errors, for example due to poor naming choices, or in parameter passing where the correspondence is positional.

Prior research aimed at enforcing dimension checking in programming language usually proposes addition of syntactic and type-checking support for some form of units and/or dimensions to the language, and expects programmers to annotate their programs with extra information [1, 10, 19]).

The lack of adoption of dimension checking proposals in mainstream programming languages indicates that a different approach is needed.

1.2 Automatic Dimension Inference

We believe that automatic inference of specifications is essential in the real world, given that it is tedious to annotate programs, and is unlikely to happen on a mass scale for routine software development. Most software projects today start from an existing code base and/or use existing libraries. This suggests that automatic inference on existing software might be a way to assimilate dimension checking into the software development process.

We have built a fully automatic tool called UniFi, short for Unit Finder, that performs dimension inference, and uses the results to detect errors. Without requiring programmer annotations, UniFi infers relations between dimensions of variables. For example, it can infer that for a program to be dimensionally consistent, two variables must have the same dimension, or that one has a dimension that is the reciprocal of the other. UniFi uses this information to automatically check for bugs introduced as a program evolves, by comparing the dimensional relationships in a new version of a program with the original ones. By reporting only the differences, UniFi can alert users to potential errors with a small false-positive rate.

Our approach has other uses besides finding errors across versions of the same program. Some possibilities are to compare consistency across different applications that use a common library, across code written by an expert and a novice, across different implementations of the same interface, or across the implementation of a library and a program that uses it. Furthermore, once dimension constraints are inferred, programmers can verify them manually and assign names to dimensions. We advocate releasing such annotations along with the interfaces of popular libraries (for example, the standard Java runtime libraries), perhaps using the JSR-308 type annotation syntax [4]. This can provide useful documentation for a large number of users, and allow a simple checker to catch dimension errors.

1.3 Contributions

The contributions of this paper are as follows:

- *An automatic dimension inference algorithm.* Our algorithm automatically infers dimension relationships between variables, fields, and parameters of primitive types and strings. Denoting the dimension of a variable x as δ_x, constraints discovered by our algorithm are of the form $\delta_{x_1}^{e_1} = \delta_{x_2}^{e_2} \times \ldots \times \delta_{x_n}^{e_n}$. We formulate dimension inference as a polymorphic type inference problem. The algorithm is especially designed to capture common idioms in object-oriented programs to provide precise results. Specifically, assuming programs obey the Liskov substitution principle [12], we infer that the corresponding parameters and return values of different methods implementing the same interface have the same dimension.

- *An error detection technique that exploits discrepancies in inferred dimensions across versions.* Errors found by our tool can be very subtle and hard to diagnose. As part of this technique, we have developed ways to match program variables and compare their dimensional relationships across two different sets of inferences.

- *Validation with experimental results.* We have implemented the proposed algorithm for Java and used it to detect bugs on a 19,000-line code base as it evolved over 10 months. The system finds two subtle errors that were independently discovered and fixed by the programmer; had UniFi been deployed on this project, the errors would have been discovered right away.

While the focus of the work reported in this paper is on the Java programming language, the general methodology and algorithms can easily be adapted to other programming languages, whether object-oriented or not.

1.4 Paper Organization

The rest of the paper is organized as follows. We first explain the dimension inference problem and show how we map it to polymorphic type inference in Section 2. Section 3 describes our algorithm to find discrepancies between dimensional constraints in different versions of a program. Section 4 describes our experience in using UniFi on a case study, and Section 5 discusses some insights we gained while using UniFi. We describe related work in Section 6, mention areas for future work in Section 7, and conclude in Section 8.

2 Polymorphic Dimension Inference

Type checking has been well established as a technique for catching various kinds of errors at compile time. While class objects often have elaborate type safety mechanisms, variables of primitive types and strings do not, though their types can also be further differentiated according to their dimensions. The hypothesis of this paper is that it is possible to build a useful checker that infers relations between the dimensions of variables automatically from the way the variables are used.

Based on the usage of variables in a program, our algorithm can infer the dimension relations between variables,

RULE	PROGRAM STATEMENT	INFERENCE
LOAD	$x_1 = x_2.f;$	$\delta_{x_1} = \delta_f$
STORE	$x_1.f = x_2;$	$\delta_f = \delta_{x_2}$
ASSIGN	$x_1 = x_2;$	$\delta_{x_1} = \delta_{x_2}$
NEW VARIABLE	$l : x = new\ C[e_1][e_2], \ldots, [e_n];$	$\delta_x = \delta_{l_1}$ $\delta_{\|l_i\|} = \delta_{e_i}, i = 1, \ldots, n$ $\delta_{l_i[]} = \delta_{l_{i+1}}, i = 1, \ldots, n-1$
ARRAY INDEX	$x = a[e];$ $a[e] = x;$	$\delta_x = \delta_{a[]}$ $\delta_e = \delta_{\|a\|}$
ARRAY LENGTH	$x = a.length;$	$\delta_x = \delta_{\|a\|}$
ADD/SUBTRACT	$x_1 = x_2\ op\ x_3;$ (*op* is $+$ or $-$)	$\delta_{x_1} = \delta_{x_2}$ $\delta_{x_1} = \delta_{x_3}$
COMPARE	$x_1\ op\ x_2$ (*op* is one of $<, >, ==, >=, <=$)	$\delta_{x_1} = \delta_{x_2}$
NEGATE	$x_1 = -x_2;$	$\delta_{x_1} = \delta_{x_2}$
REMAINDER	$x_1 = x_2\ \%\ x_3$	$\delta_{x_1} = \delta_{x_2}$
MULTIPLY	$x_1 = x_2\ \times\ x_3;$	$\delta_{x_1} = \delta_{x_2} \times \delta_{x_3}$
DIVIDE	$x_1 = x_2\ /\ x_3;$	$\delta_{x_1} = \delta_{x_2} \times \delta_{x_3}^{-1}$

Table 1. Intraprocedural inference rules used by UniFi

but not their absolute semantics. For example, if two variables are added, we can infer that the variables must have the same dimension, though we do not know a precise name for that dimension. Note that a variable can also be inferred to be dimensionless, i.e. a pure number. For example, if a program contains a statement such as $n = n^2$ or $m = m/n$, then n must necessarily be dimensionless for the program to be well-typed.

In general, it is possible for dimensions to form a hierarchy, in a way that is analogous to a class hierarchy. For example, it may be legal to add the number of apples to the number of oranges, if the result is supposed to be the number of fruits. However, in this paper, we assume a flat hierarchy for dimensions, i.e., there are no subtyping relationships between dimensions. This choice simplifies our analysis and keeps the number of false-positive warnings low.

2.1 The Program Model

Our inference algorithm is designed to find dimension relationships between scalar and array variables of strings and primitive types in Java: `int`, `long`, `byte`, `char`, `short`, `boolean`, `float` and `double`. Though it handles the full Java language, for the sake of brevity, we will use a simplified language as shown in Table 1. The table provides a summary of the intraprocedural inference rules; how we handle methods is described in Section 2.3.

We model dimensions in a program as follows:

- Each local variable x has dimension δ_x. Every use of a constant is treated as a separate local variable. (As a preprocessing step, we rename logically different variables that happen to use the same local variable slot, using an analysis similar to deriving the static single assignment form [16]).
- Fields, static or instance, are monomorphic. Field f of all instances of a class are assumed to have the same dimension δ_f.

- All objects created at allocation site l are treated as having the same dimension δ_l. A multi-dimensional array is modeled as an array of arrays. The dimension of an array a, δ_a, has two related components:
  1. Dimension of the elements, $\delta_{a[]}$. All elements in an array are assumed to have the same dimension.
  2. Dimension of the length of the array, $\delta_{|a|}$.
- Methods can be polymorphic, allowing parameters in different invocations to take on different dimensions. The dimension summary of method m is expressed as constraints between class fields and parameters $\delta_{m[i]}$, where $\delta_{m[i]}$ represents the the dimension of the ith parameter, and $\delta_{m[0]}$ is the dimension of the return value.

2.2 Intraprocedural Inference Rules

Loads, stores, and assignments generate unification constraints between the dimensions of the source and destination variables. For array element access, the dimension of the array index is unified with the length of the array. This can be useful to detect bugs where a dimensionally incorrect variable is used to index an array.

Addition, subtraction, and comparison are all operators whose operands are expected to have the same dimension, whereas addition, subtraction and negation produce results of the same dimension as the operands. The result of a remainder operator shares the same dimension as its first operand.

Multiply and divide operations are special from a dimensional perspective because they produce results with composite dimensions. We refer to such constraints as *composite dimensional constraints*. Given a statement $x_1 = x_2 \times x_3$, we can infer that $\delta_{x_1} = \delta_{x_2} \times \delta_{x_3}$. Though not detailed in Table 1 for purposes of brevity, UniFi also handles the semantics of the `java.lang.Math` library methods such as `sqrt` (the dimension of the return value is the square root of the parameter), and `abs`, `floor`, `round`, `min`, `max` (the dimension of the return value is the same as the parameter). The `pow` method generates the appropriate constraint if the exponent is a simple compile-time rational constant, otherwise it is ignored.

Our dimension inference algorithm often provides constraints we would normally associate with units, rather than dimensions. (Units are multiple scales for measuring the same dimension, e.g. *foot* and *meter* are both units of the dimension *length*.) Given the program statement:

```
yen = dollars * exchangeRate;
```

our analysis infers that the dimension of `exchangeRate` is the dimension of `yen` divided by the dimension of `dollars`. If we later encounter an erroneous statement:

```
dollars = yen;
```

the analysis will infer that `yen` must have the same dimension as `dollars`, and `exchangeRate` must therefore be dimensionless. As will be discussed in Section 3, a change in dimensions across versions causes UniFi to raise a warning of a potential error.

A popularly cited example related to units errors is the crash of the $125 million Mars Climate Orbiter due to a failure to convert between metric and English units in software [13]. However, not all unit errors can be caught using our approach, especially in cases where the wrong scaling factor is used.

2.3 Interprocedural Analysis

In the following, we first describe an additional inference rule for methods, and then the inference algorithm itself.

2.3.1 Subtyping Constraints

The Liskov Substitution Principle in object-oriented programming requires that the contract of a method remain the same in all subtype implementations [12]. This implies the constraint that the dimensions associated with each method's corresponding parameters and return value remain the same as well. A stricter form of this constraint would be that a method in a subtype must have covariant parameters and contravariant return types. However, since we use a flat hierarchy, we simply require that the parameters and return values have the same dimension.

In practice, we can just substitute a reference to method $T.m$ with a reference to $T'.m$ where T' is the most generic class or interface of T that contains the same method interface m. If there are multiple most generic interfaces, then we generate constraints to unify the respective parameters and return values of m in all those interfaces. The constraints inferred due to each method body implementing a given interface all update the constraints at the interface.

2.3.2 Polymorphic Dimension Inference

From our early attempts of using a monomorphic dimension inference algorithm, we have found programs to have at least a few dimensionally polymorphic methods. Thus, it is important to have a context-sensitive dimension inference algorithm to avoid conflating dimensions from different call sites to polymorphic methods.

To achieve context sensitivity, our algorithm summarizes the effect of methods with dimensional constraints between input parameters and return values, and other global dimensions. These method summaries can involve unification as well as composite dimensional constraints. These summaries are applied at each call site. Our algorithm computes

the summaries of the methods iteratively until convergence is reached.

2.4 UniFi Usage

A user supplies each run of UniFi with a set of Java class files. Class files may be compiled with or without debug information; if debug information is present, it is used to print accurate local variable and method parameter names. Since UniFi analysis runs directly on bytecode, the class files analyzed could potentially be generated from multiple source languages like Java or Java Server Pages. Methods in external libraries are treated as black boxes unless explicitly included in the set of classes to be analyzed.

With every variable, UniFi tracks all the constraints associated with it, and with every constraint, it tracks the point in the code that created the constraint. This information is essential for explaining intelligibly to the user why UniFi inferred the dimensional constraints that it did.

The UniFi GUI lets users graphically browse the results of an analysis run. The user can view all the inferred dimensions, and the variables in each one of them. The GUI also correlates constraints with source code and lets the user query why two variables were inferred to have the same dimension.

3 Comparing Inferences Across Programs

We now describe how UniFi compares two sets of dimensional constraints inferred by the algorithm described in the previous section. There are three aspects to this comparison. The first is to identify common dimension variables in the two sets. The second is to check whether these variables form the same equivalence classes. The third is to check whether composite dimensional constraints are equivalent in the two versions.

3.1 Identifying Common Variables

To enable comparison of inference results across two different programs or two versions of the same program, we first need to find correspondence between variables across the two programs. A simple heuristic that we adopt in our current implementation is the following. We match fields with the same fully qualified name. We match method parameters, return values and local variables by the full method signature and position of the parameter or local variable. More robust association mechanisms that allow, for example, for systematic re-factoring of code are possible.

3.2 Comparing Unification Constraints

Once we have identified variables that are common in the two program versions, we check if the dimensions for these variables form the same equivalence classes in both programs. If not, this means that some set of variables that were in different equivalence classes in one set of results were unified in the other; this is reported as a potential dimensional consistency violation. We initially expected that errors would be detected with unification constraints mainly when dimensions that used to be previously independent in a program were subsequently unified. However, as described later in Section 4, we also uncover errors when dimensions that were in the same equivalence classes became independent in a subsequent version.

When a dimension error is reported, we find it useful to let users query in our GUI the smallest set of unifications that caused two variables to share the same dimension. The user can investigate these unifications and correlate each unification with the point in the source code that caused it.

3.3 Comparing Composite Dimension Constraints

Recall that the constraint generation phase sets up a system of composite dimension constraints involving different dimensions. UniFi converts each constraint to the form:

$$\delta_1{}^{e_1} \times \delta_2{}^{e_2} \times \ldots \times \delta_n{}^{e_n} = 1$$

where $\delta_1, \ldots, \delta_n$ are all the dimensions in the program and each exponent e_i is a rational number. For example, the statement $E = m \times c^2$ is converted to the constraint:

$$\delta_E \times \delta_m{}^{-1} \times \delta_c{}^{-2} = 1.$$

It is useful to reduce this system of constraints to a canonical form, in order to enable simple comparison of constraints across the two versions of the program. A second benefit is that we can ensure that the results of the analysis can be presented to programmers consistently, without being perturbed by extraneous issues like the order in which program statements are processed.

We derive a canonical form for composite dimension constraints by expressing dependent dimensions in terms of other, independent dimensions. Independent dimensions are just those that are not expressed in terms of others (similar to the fundamental S.I. units in physics). However, our choice for selecting which dimensions to consider independent can be somewhat arbitrary. For example, the constraint:

$$\delta_E \times \delta_m{}^{-1} \times \delta_c{}^{-2} = 1$$

could result in the derivation of any of the following equations, based on which two of the three dimensions involved are considered independent:

$$\delta_E = \delta_m \times \delta_c^2$$
$$\delta_c = (\delta_E/\delta_m)^{1/2}$$
$$\delta_m = (\delta_E/\delta_c^2)$$

We call the expression for a dependent dimension δ_v in terms of a composition of other, independent dimensions a formula for δ_v.

To derive canonical formulas for dependent variables, we first define a priority order for selection of independent dimensions. Elements with higher priority are preferred as independent dimensions to elements with a lower priority. Our goal is to derive a set of formulas for dependent variables in terms of other independent variables that are all of higher priority.

As the first step, we rewrite each constraint, replacing each dimension with the highest priority dimension in the same equivalence class. Next, we reduce our system of constraints to canonical formulas using a Gaussian elimination style algorithm. The algorithm is described in Figure 1. It takes in a set of constraints and a priority ordering. It successively eliminates lower priority dimensions from the constraint system by replacing them with formulas composed of higher priority dimensions. It then back-substitutes each of the dimensions in each formula, this time going from the highest priority dimension downwards. This ensures that dependent dimensions are expressed as a function of independent dimensions.

To compare two sets of composite dimension constraints, we adopt a priority order for independent dimensions in each program such that the dimensions of variables in common are ordered after those not in common. The common dimensions are also sorted to ensure they have a consistent order for both constraint systems. We then use the algorithm in Figure 1 to generate formulas in each set of constraints. The formulas for the dependent dimensions that are common are expressed in terms of other common dimensions when possible. Given these canonical formulas in both versions, it is simple to check whether the formulas for the common dependent variables are the same in both versions. If they are different, a possible dimension error is flagged. An error can also be flagged if a variable that has a non-empty dimensional formula in one version is flagged as dimensionless in the other.

Inputs:

1. A list of dimensions D: $\delta_1, \ldots, \delta_n$ in increasing order of priority.
2. A set of constraints $C = \{c_1, c_2, \ldots, c_m\}$, where c_i is
$$\prod_{1 \le j \le m} {\delta_j}^{e_{ij}} = 1$$
and each exponent e_{ij} is a rational number.

Outputs:

1. A set of dependent dimensions $D' \in D$
2. A set of constraints $C' = \{c'_i \mid \delta_i \in D'\}$, where c'_i is $\delta_i = F_{\delta_i}$, and F_{δ_i} is of the form ${\delta_{i+1}}^{e_{i+1}} \times \ldots \times {\delta_n}^{e_n}$.

Algorithm:

```
for j = 1 to n do
    pick some c_i ∈ C such that e_ij ≠ 0
    if no such c_i exists
        continue to the next j
    F_δj ← (∏_{1≤k≤n,k≠j} δ_k^{e_ik})^{-1/e_ij}
    remove c_i from C
    foreach c ∈ C do
        rewrite c replacing δ_j with F_δj
        update exponent matrix e to reflect the exponents in the
        rewritten constraint
for j = n down to 1 do
    if F_δj is defined
        rewrite F_δj replacing δ_k with F_δk,
        where k = j + 1, ..., n and F_δk is defined
        add δ_j to D' and the constraint δ_j = F_δj to C'
```

Figure 1. Algorithm to canonicalize composite dimension constraints

4. UniFi Case Study

In this section, we report results of running UniFi over the codebase of bddbddb, an open source program analysis toolset. This toolset is implemented in Java and is hosted at the public open source repository SourceForge. We ran UniFi retrospectively over daily snapshots across 10 months (from October 2004 to July 2005) when this project was under active development. We chose this project because we knew this project had evolved significantly during this period and we had easy access to the developer of the project for analyzing the results generated by UniFi.

We built the main trunk of the repository as it existed each night of this period. There were a total of 292 successful builds. For each successful build (except the first), we compared UniFi results with the results from the previous build and reported differences in dimensional relationships. All dates in this section are in Year-Month-Date format.

4.1 Codebase Statistics

Table 2 provides statistics about this codebase as of 2004-10-01 and as of 2005-07-30. Typical analysis runtime for each inference run was between 20 and 25 seconds on a 2.2GHz Intel CPU with 4 GB physical memory, and a 512MB JVM heap size. The row listing "number of distinct method interfaces" is different from the number of method bodies analyzed because multiple methods could map to the

	2004-10-01	2005-07-30
Classes	179	226
Interfaces	14	14
Method bodies	1,376	1,801
Distinct Method interfaces	889	1215
Binary jar file size (KB)	359	463
Lines of code (Non-blank, non-comment)	14,379	19,119

Table 2. Codebase statistics

Type of Dimension Variable	Count
Field	258
Local variable	877
Constant	2,244
Method parameter	552
Method return value	393
Result of multiply/divide	102
Array element	129
Array length	407
Total	4,962

Table 3. Frequency of dimension variables

same interface method, using the object-oriented interface mappings explained earlier.

Table 3 lists the number of dimension variables in the program from date 2005-07-30. Note that these counts consider only variables of primitive types and strings. For this program version, the largest equivalence class was of size 132. There were 3340 equivalence class containing just one element; of these 2100 were one-time use constant strings.

4.2 Bugs During Software Evolution

In all, over 292 builds, comparing the results of each build with the results on the previous one caused 26 warnings between 19 pairs of builds. We have found that the most interesting reports tend to be those where there is a change in the dimensional relationships of a (static or instance) field or a method interface (method parameter or return value) — these are typically more useful than reports involving only, say, local variables. We are considering restricting UniFi reports to those that involve one of these kinds of variables. If we consider only fields and method interfaces, there were only 16 warnings. 5 of these 16 warnings were dimensional differences directly related to bugs: 2 introduced new bugs, and 3 fixed bugs. The other 11 warnings were caused by valid changes in dimensional relationships as the program evolved.

We describe below the three errors encountered by UniFi in bddbddb. Of these three, the first two would have been found by UniFi had it been deployed during development. All three errors were independently discovered by the developer and fixed at some point after they were introduced.

Bug 1: A code update on 2004-11-07 caused UniFi to issue a warning saying that the previously independent dimension variables associated with the fields `NO_CLASS_SCORE` and `NO_CLASS` in the class `FindBestDomainOrder` (in package `net.sf.bddbddb`) were now unified. The field `NO_CLASS` has the same dimension as a group of variables referring to a class identifier; `NO_CLASS` is used as a special default value. The `NO_CLASS_SCORE` field has the same dimension as the score of a class, which is unrelated to a class identifier; once again, `NO_CLASS_SCORE` is a special default value for the score of a class.

In method `tryNewGoodOrder2` in this class, the following initialization code was introduced:

```
...
double vScore=NO_CLASS,aScore=NO_CLASS,
dScore=NO_CLASS;
double vClass=NO_CLASS,aClass=NO_CLASS,
dClass=NO_CLASS;
...
```

The intention here is clearly to set the score variables to a default score value of `NO_CLASS_SCORE`; however, they are incorrectly set to `NO_CLASS`. One can imagine that it is easy to make such a mistake since both variables have similar names and are declared and used in the same area of the code. This error is fixed by an update on 2004-11-11, at which point UniFi reported another warning saying that the previous unified dimension variables for `NO_CLASS` and `NO_CLASS_SCORE` were now independent.

Bug 2: Somewhat surprisingly, UniFi can also finds errors when variables that should be unified are separated into different equivalence classes. The following example illustrates this: Analysis on the code as of 2005-05-24 reported that the previously unified dimension variables for the fields `net.sf.bddbddb.Stratify.TRACE` and `net.sf.bddbddb.Solver.TRACE` were now independent. The original code was the following:

```
public Stratify(Solver solver) {
    this.solver = solver;
    this.TRACE = solver.TRACE;
    this.out = solver.out;
}
```

This constructor in the `Stratify` class which takes a `Solver` as a parameter and copies the `Solver`'s `TRACE` field to its own `TRACE` field. This code was changed on 2005-05-24, but inadvertently lost the copy of this field:

```
public Stratify(Solver solver) {
    this.solver = solver;
    this.NOISY = solver.NOISY;
    this.nodes = new HashMap();
    this.emptyRelationNode =
        getRelationNode(null);
}
```

The copy of `TRACE` was re-introduced in a bug fix a few days later on 2005-06-11.

Bug 3 (not detected by UniFi): This bug illustrates a limitation of UniFi: a dimension error was introduced in new code. Since UniFi implicitly assumes dimensions associated with variables that it has not seen before to be correct, it did not generate a warning. Manual inspection of dimensions associated with new variables may therefore be useful for detecting these kinds of errors. Our attention was focused on this example because UniFi correctly detected the change in dimension relationships when the bug was fixed. The error was as follows:

The class `net.sf.bddbddb.order.BaggedId3` has 2 fields: `numClasses` and `NUM_TREES`, which have logically different dimensions. The dimensions for these fields were being incorrectly merged due to the following loop in method `distributionForInstance()`:

```
double[] distribution =
    new double[numClasses];
... //compute sum and initialize
... // distribution array
for (int i=0; i<NUM_TREES; ++i)
    distribution[i] /= sum;
```

The for loop above should have run for the range [0..`numClasses`-1] instead of [0..`NUM_TREES`-1]. The effect of this bug was to unify the dimensions of `numClasses` and `NUM_TREES` via the variable `i` and the length of the `distribution` array; the bug fix correctly caused UniFi to report that these variables were now in different equivalence classes.

This bug was introduced in new code on 2004-11-09; it was fixed on 2004-11-11.

In our experiments with UniFi on bddbddb, no dimensional difference reports were issued which involved compound constraints due to multiply/divide operations; all reports were related to unification of dimension variables. This probably reflects the fact that bddbddb does not perform many multiply/divide operations, except a few for statistical reporting.

5 Discussion

In this section, we discuss our overall experience in using UniFi on bddbddb as well as on other projects.

Types of Errors: Many errors due to dimensional inconsistency tend to be relatively simple errors, similar to the errors found by conventional type checking. However, we have also come across situations where a dimensional error has taken several days to debug. This is typical, for example, when the range of the incorrect value is similar to that of the correct value, such as a small integer. The use of the incorrect value may not lead immediately to an obvious failure like a crash, and may even cause silent data corruption.

False Positives: We have not yet made significant efforts to reduce the false positive rate because, in practice, the absolute number of false positives is small and has not been problematic. Running on nightly builds, UniFi typically issues a warning only once in a few days, and even when the report does not point out a bug, it highlights new relationships between variables in the code in interesting ways. One reason we see for false positives is when a field is declared in a class, but its functionality is initially left unimplemented. When the field is eventually used, UniFi emits a warning because the dimension of the field merges with other dimensions in the program.

Another reason for false positives is when a constant primitive type field switches between being declared final and non-final. A final declaration causes the javac compiler to remove the reference to the field and replace its use with a compile time constant, making the field itself unused at the bytecode level. We expect that it should be fairly easy to eliminate or de-prioritize these kinds of false positives.

A more interesting source of false positives (or lost precision) is due to our treatment of all elements of an array having the same type. Some coding patterns, most notably those involving reflection, assemble different kinds of variables (e.g. strings) into a single array, thereby merging distinct dimensions. A possible workaround for this specific pattern is to try and assign different dimension variables if an array is only accessed with constant indices.

Dimensionally-inconsistent code: We find that some methods, most notably those that override `Object.hashCode()` or implement `Comparable.compareTo()`, intentionally perform dimensionally inconsistent computations. A common idiom for `hashCode()` is to compute an arbitrary function of the object's fields. A common idiom for `compareTo()` is the following:

```
public int compareTo (Object o) {
  SomeClass other = (SomeClass) o;
  if (this.field1 != other.field1)
     return (this.field1 - other.field1);
 return this.field2 - other.field2;
}
```

This idiom compares two objects of a type by first comparing a major field (`field1`) and then comparing a minor

field (`field2`) if the major fields are equal. However it conflates the dimensions of `field1` and `field2` by unifying both with the dimension of the return value of the `compareTo` method. Therefore, UniFi always excludes method bodies for both of these special methods from its analysis.

Variable Naming: One side effect we have observed during analysis of UniFi results is that browsing the variables with a given dimension sometimes points to a poor choice of variable names. While generic local variable names like *i* and *n* commonly appear in different dimension classes, we also see cases where a variable's dimension is correct, but the variable is named misleadingly, reflecting confusion in the mind of the programmer. Poor naming can mislead the reader of the code about the semantics of the variable and cause a potential software maintenance problem.

6 Related Work

UniFi belongs to a general class of tools that attempt to acquire specifications automatically by mining existing software, either statically or dynamically. These approaches exploit the property that programs are often mostly correct, and can thus be a useful source of specifications. The inferred specifications can be used to check for errors in many ways: for example, by detecting inconsistencies within the specifications, verifying the code statically against the specifications, or checking for violations of invariants in dynamic runs of the program [2, 5, 9, 11, 20]. In UniFi's case, we attempt to use existing code to derive program specific dimensions and the relationships between them.

Prior approaches to enforcing dimensional consistency in software depend on programmers providing annotations or modifying programs in some way. Fortress is a research programming language from Sun Microsystems that provides support for units and dimensions in an object-oriented setting by extending the syntax and semantics of the Java programming language [1]. Van Delft proposes another extension to Java to support dimensions [19]. The Xelda system checks dimensional correctness of spreadsheets and found bugs in several scientific computing spreadsheets accompanying a textbook [3]. Osprey is a type-checking system for C that tries to limit the programmer burden by requiring annotations on some set of variables, but inferring dimensions on others [10]. However, Osprey is limited to checking dimensions that are a function of a fixed set of units, like the S.I. units.

Like UniFi, the Fortress, Osprey and Xelda systems mentioned above use abelian groups to represent dimensional algebra constraints related to multiplication and division, and employ techniques similar to Gaussian elimination to solve such constraints. Unlike UniFi, which can potentially be used automatically to detect dimension errors, these three systems depend on some form of user annotation to seed the type system. Further, only Fortress exploits subtyping relationships in object-oriented languages.

Lackwit is a tool that performs polymorphic type inference on a C program to identify variables that are constrained to have the same representation [14]. This information is used as an aid to performing software maintenance. However, Lackwit does not generalize its type inference to support dimensions and does not support arithmetic operators like addition and multiplication.

Some prior work attempts to infer dimensional consistency at runtime. Guo et al attempt to infer abstract types by instrumenting a program and detecting interactions at run time [8]. Their built-in interactions do not handle multiply and divide constraints using abelian groups. Petty's proposed approach for Fortran aims to ensure dimensional consistency (mainly for the S.I. units) by using a modified real type to carry dimensional information [15].

Another body of related work uses the concept of type qualifiers. The Cqual and Jqual frameworks (for C and Java programs respectively) provide generalized type inference and checking using type qualifiers. They let programmers assign qualifiers to type declarations, and describe how the type qualifiers interact with the operators of the language [6, 7]. UniFi's dimensions can be viewed as a particular class of type qualifiers, although UniFi does not support subtyping relationships between dimensions. However, UniFi targets the specific domain of dimensions, and therefore embeds constraint rules and solvers specific to this domain. Uses of Cqual so far have been in the domain of constant variable inference [6] and taint propagation[17]; Jqual has been applied to enhance type checking with respect to native (JNI) code and for detecting immutable variables. JavaRI is another system that attempts to use type checking to verify immutability properties of annotated Java programs [18]. UniFi's general approach of inferring dimensions on one version of the program and using the results to check other versions of the program may be useful with other kinds of type qualifiers as well.

7 Future Work

As mentioned earlier, there are many different situations in which it may be useful to compare dimension inference results. More work is needed to gain experience with UniFi in these situations. More experience is also needed with scientific applications where there is an abundance of multiply and divide relationships.

One promising approach is to use UniFi to infer dimensions on the implementation of popular Java libraries. After manual review and assignment of human-friendly names, the inferred dimensions can be output as type annotations

using the JSR-308 syntax. This will offer a considerable amount of documentation to users of these libraries and a way for compile time checkers to detect dimension errors.

We could also extend UniFi to precompute summaries of methods in popular libraries, so that the effect of these methods can be accurately applied to programs that invoke them, without the need to analyze the libraries along with every program.

Finally, we are interested in exploring dimensional consistency in the context of hardware programs written in languages like Verilog. Hardware programs have even less protection than software in terms of type checking, and inferring dimensions in a sea of bits may be a useful way to find inconsistencies in the design.

We plan to release our UniFi implementation in open source form. Additional information and screenshots of the UniFi GUI are available at the website: http://cs.stanford.edu/~hangal/unifi.html.

8 Conclusions

We have shown that the UniFi approach is a practical way to bootstrap the use of dimensional analysis into the software development process. Dimension checking is useful for much more than scientific code; it is valuable in all types of code because programs manipulate many different kinds of values that have dimensions associated with them.

9 Acknowledgments

We thank Rajit Badgandi for working on early parts of the UniFi implementation, John Whaley for letting us use bddbddb as a test case and providing us feedback on the bugs, Christopher Unkel for useful discussions and the anonymous reviewers for valuable feedback. This work is supported in part by the National Science Foundation under TRUST grant #0424422 and a Stanford graduate student fellowship.

References

[1] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, and G. L. Steele, Jr. Object-oriented units of measurement. In *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 384–403. ACM, 2004.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16. ACM, 2002.

[3] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the unit correctness of spreadsheet programs. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 439–448. IEEE Computer Society, 2004.

[4] A. Buckley and M. D. Ernst. Java Specification Request-308: Annotations on Java Types. *http://jcp.org/en/jsr/detail?id=308*.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 57–72. ACM, 2001.

[6] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 192–203. ACM, 1999.

[7] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 321–336. ACM, 2007.

[8] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 255–265, July 18–20, 2006.

[9] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301. ACM, 2002.

[10] L. Jiang and Z. Su. Osprey: a practical type system for validating dimensional unit correctness of c programs. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 262–271. ACM, 2006.

[11] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 161–176. USENIX Association, 2006.

[12] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[13] T. Mars Climate Orbiter Mishap Investigation Board. Phase 1 Report *ftp://ftp.hq.nsa.gov/pub/pao.reports/1999/MCO_report.pdf*.

[14] R. O'Callahan and D. Jackson. Lackwit: a program understanding tool based on type inference. In *ICSE '97: Proceedings of the 19th International Conference on Software Engineering*, pages 338–348. ACM, 1997.

[15] G. W. Petty. Automated computation and consistency checking of physical dimensions and units in scientific programs. *Software: Practice and Experience*, 31(11):1067–1076, 2001.

[16] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM, 1988.

[17] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *SSYM'01: Proceedings of the 10th Conference on USENIX Security Symposium*. USENIX Association, 2001.

[18] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, October 18–20, 2005.

[19] A. van Delft. A Java extension with support for dimensions. *Software: Practice and Experience*, 29(7):605–616, 1999.

[20] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 218–228. ACM, 2002.