Representation Dependence Testing Using Program Inversion

Aditya Kanade
Indian Institute of Science, Bangalore

Rajeev Alur
University of Pennsylvania, alur@cis.upenn.edu

Sriram Rajamani
Microsoft Research India

Ganesan Ramalingam
Microsoft Research India

Follow this and additional works at: http://repository.upenn.edu/cis_papers

Part of the Computer Sciences Commons

Recommended Citation
Representation Dependence Testing Using Program Inversion

Abstract
The definition of a data structure may permit many different concrete representations of the same logical content. A (client) program that accepts such a data structure as input is said to have a representation dependence if its behavior differs for logically equivalent input values. In this paper, we present a methodology and tool for automated testing of clients of a data structure for representation dependence. In the proposed methodology, the developer expresses the logical equivalence by writing a normalization program $f$ that maps each concrete representation to a canonical one. Our solution relies on automatically synthesizing the one-to-many inverse function of $f$: given an input value $x$, we can generate multiple test inputs logically equivalent to $x$ by executing the inverse with the canonical value $f(x)$ as input repeatedly. We present an inversion algorithm for restricted classes of normalization programs including programs mapping arrays to arrays in a typical iterative manner. We present a prototype implementation of the algorithm, and demonstrate how our methodology reveals bugs due to representation dependence in open source software such as Open Office and Picasa using the widely used image format Tiff. Tiff is a challenging case study for our approach.

Disciplines
Computer Sciences
Representation Dependence Testing using Program Inversion

Aditya Kanade
Indian Institute of Science
kanade@csa.iisc.ernet.in

Rajeev Alur
University of Pennsylvania
alur@cis.upenn.edu

Sriram Rajamani  G. Ramalingam
Microsoft Research India
{sriram,grama}@microsoft.com

ABSTRACT

The definition of a data structure may permit many different concrete representations of the same logical content. A (client) program that accepts such a data structure as input is said to have a representation dependence if its behavior differs for logically equivalent input values. In this paper, we present a methodology and tool for automated testing of clients of a data structure for representation dependence. In the proposed methodology, the developer expresses the logical equivalence by writing a normalization program \( f \) that maps each concrete representation to a canonical one. Our solution relies on automatically synthesizing the one-to-many inverse function of \( f \): given an input value \( x \), we can generate multiple test inputs logically equivalent to \( x \) by executing the inverse with the canonical value \( f(x) \) as input repeatedly. We present an inversion algorithm for restricted classes of normalization programs including programs mapping arrays to arrays in a typical iterative manner. We present a prototype implementation of the algorithm, and demonstrate how our methodology reveals bugs due to representation dependence in open source software such as Open Office and Picasa using the widely used image format Tiff. Tiff is a challenging case study for our approach.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification; D.2.5 [Software Engineering]: Testing and Debugging; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms

Experimentation, Reliability, Verification

Keywords

Data Structures, Program Inversion, Testing

1. INTRODUCTION

In this paper, we address data representation dependence, a class of bugs that is a common source of incompatibility between programs. We describe a new testing technique that we call representation fuzzing to identify such bugs.

Data Representation Dependence. Consider a program (the producer) that produces some data that is consumed by another program (the client). When the client program works correctly with one version of the producer, but fails to do so with a latter version of the producer, we have a version compatibility problem.

Data definition is often a key to interoperability between programs. By “data definition”, we mean a definition of the structure of data interchanged between programs and its semantics (i.e., the data value a particular data instance represents). Such a definition essentially acts as the contract between a producer and client of data.

Such definitions often permit different representations of the same data value. A new version of a producer program may exploit the flexibility in the data definition contract to produce output that looks structurally different from the output of an earlier version of the producer, yet semantically represents the same value. A correct client is expected to work with the output of both versions of the producer.

Unfortunately, an implementation of a client may make assumptions about the data definition based on the observed output of a particular version of the producer program rather than use an independently specified data definition. As a result, it may fail to work correctly when the implementation of the producer evolves. We refer to such a client as having a data representation dependence. Using examples of clients using Windows data structures and clients using the widely used image format Tiff, we argue that such dependence is indeed an indicator of bugs in practice.

Representation Fuzzing. In this paper, we propose an effective novel technique for testing a client program \( C \) for representation dependence bugs. Given a test case (concrete input) \( x \), we first (automatically) produce other test cases \( z \) that are logically equivalent to \( x \). We then test the program \( C \) with these different, but equivalent, inputs. If the program exhibits different behaviors for different, but equivalent, inputs, we have a representation dependence bug. Differing behaviors could take the form of successful completion in one case and a failure (e.g., null pointer dereference) in the other, or different (inequivalent) outputs in the two cases.

This technique, which we call representation fuzzing, works by essentially finding alternative representations of a test case without altering its semantics. In our methodology, the
bounded arrays, and indirectness in array indexing and is a challenging case study for inversion. We cover strip-based TIFF images with flexibility in the choice of orientation, rows per strip, and physical layout of strips. We test five popular TIFF clients, namely, Open Office, KView, GIMP, FastStone, and Picasa with the following conclusions:

- Even if the number of rows per strip is varied, all clients process the image correctly for all test cases.
- If orientation is varied then Open Office and GIMP display images incorrectly with no warning to the user.
- If logically adjacent strips are reordered physically in conjunction with change in orientation then Picasa displays images incorrectly with no warning to the user.

2. PROPOSED METHODOLOGY

We now formulate the problem concretely with an example of representation dependence and by specifying syntax and semantics of normalization programs and their inverses.

In a programming language like C, the concrete representation of data can be given as a struct declaration. As an example, consider the following simplified version of datatype `ddsurfacedesc` in the Windows DirectDraw API:

```
1 typedef struct _ddsurfacedesc {
2   dword height, width, pitch;
3   lpvoid surface;
4 } ddsurfacedesc;
```

A data `d` of type `ddsurfacedesc` contains two dimensional pixel data `d.surface`. We assume that a pixel is stored as a single byte. The number of rows and columns of the pixel data are given by `d.height` and `d.width`. The field `d.pitch` gives the number of bytes between a pixel and the pixel immediately below it. A program can append `slack bytes` to each row of the pixel data for internal use. The value of `(d.pitch - d.width)` is intended to indicate how many slack bytes are used by a program. In practice, slack bytes may not be used by a producer program `P`.

Consider the following client program `C` which uses data `d` of type `ddsurfacedesc` as input and reads the `c`th column of `d.surface` into an array `col`.

```
1 for i := 1 to d.height do
2   col[i] := d.surface[d.width * i + c];
3 end
```

It works correctly on data generated by `P`. However, if `P` is changed `independently` to use slack bytes then it reads the pixels incorrectly. In other words, `C` makes a stronger assumption on the datatype (pitch = width) than the specification allows (pitch ≥ width) and it fails if pitch > width.

![Figure 1: Automated Testing for Representation Dependence (of Clients of TIFF Images)](image-url)
This example of representation dependence is motivated by a real bug involving a client of ddsurfacedesc described in the Windows application compatibility bug database (cf. [4]).

In order to detect representation dependence of clients of a datatype $T$, we need a formal specification of the equivalence relation $\approx$ over $T$. If a client program $C$ produces different behaviors or outputs on data $d$ and $d'$ of type $T$ such that $d \approx d'$ then the client has representation dependence on $T$. For ddsurfacedesc, $d \approx d'$ iff

- $d$.width = $d'$.width, $d$.height = $d'$.height, and
- $d$.surface[$d$.pitch * $i$ + $j$] = $d'$.surface[$d'$.pitch * $i$ + $j$] for all $i, j$ such that $1 \leq i \leq d$.width and $1 \leq j \leq d$.height.

Given a data $d$, the goal of representation fuzzing is to automatically generate $d'$ such that $d' \approx d$ and test a client with both $d'$ and $d$. Writing logical specifications like this may be impractical for the developers. Further, analyzing complex logical specifications seems difficult. Instead, we propose that the developer writes a normalization program $f$ to capture the canonical representation of a datatype $T$. This is a reasonably straight-forward task. We define $d' \approx d$ iff $f(d') = f(d)$. We then show how to use program analysis to synthesize an inverse $f'$ and obtain $d'$ by applying $f'$ to the canonical representation $f(d)$.

**Normalization Programs.** Consider a datatype nform with the fields height, width, and a two dimensional array data as the canonical representation of ddsurfacedesc. The normalization program is given below.

```plaintext
/* Input: d, Output: n */
1 n.height := d.height; n.width := d.width;
2 for i := 1 to d.height do
3     for j := 1 to d.width do
4         n.data[i][j] := d.surface[d.pitch * i + j];
5 end
6 end
```

Figure 2 gives the syntax of normalization programs. The input and output variables are indicated by letters $x$ and $y$ respectively and for simplicity, are considered to be integer valued. Program expressions (exp) and predicates (pred) are affine. Loop counters and bounds are named $i, j, k$ and $m, n$ etc. A loop counter is initialized to 1 and incremented by 1 in each iteration until the upper bound is reached. The loops are required to be perfectly nested, that is, all assignment and assume statements are part of the innermost loop body which can only be straight-line code. An array index expression can be an affine expression. To handle the TIFF case study, we address two special cases of non-linear index expressions: a scalar input variable to be a coefficient of an affine expression and a single level of indirection through another array. The index expressions of an input array can differ only in constant terms. The RHS of an assignment to an index variable (local variable) $i$ contains only $i$ and constants. We assume that an output variable $y$ is assigned exactly once in the program and is not used before it is defined. This also applies to every output array location $y[f]$. This assumption is met in the TIFF case study. Though similar to some constraints we check algorithmically for inversion, presently, we verify this assumption manually. Some of the syntactic extensions are not mentioned for brevity e.g., boolean connectives are allowed in predicates.

While the program semantics are as usual, we assume invertibility of assignment statements: An assignment statement $y := e$ is invertible with respect to every variable $x$ appearing in $e$ if the equality relation $y = e$ can be rewritten to a logically equivalent relation $x = e'$. Since we assume that an output variable is assigned exactly once and is not used before it is assigned, $y$ does not appear in $e$. The variables of $e'$ can only be from $V(e) \cup \{y\}\setminus \{x\}$ where $V(e)$ is the set of input variables appearing in $e$.

**Inverse Programs.** An inverse of the normalization program of ddsurfacedesc is given below.

```plaintext
/* Input:n, Output:d */
1 ensure(d.pitch >= n.width);
2 d.height := n.height; d.width := n.width;
3 for i := 1 to d.height do
4     for j := 1 to d.width do
5         d.surface[d.pitch * i + j] := n.data[i][j];
6 end
7 end
```

The inverse program non-deterministically assigns a value to $d$.pitch such that $d$.pitch $\geq$ $n$.width. The values of height and width fields of $d$ and $n$ are same. The control flow of the loop in the inverse program is same as that of the normalization program. It however copies $n$.data to $d$.surface. The condition $d$.pitch $\geq$ $n$.width = $d$.width (Line 1) ensures that every element of $d$.surface is assigned at most once within the loop. Thus, if an output $d$ of the inverse program on $n$ is given as the input to the normalization program, we get back the same canonical representation $n$. Suppose the inverse chooses $d$.pitch = $d$.width+1. Since $d$.pitch $>$ $d$.width, there are slack bytes but they are not required to be initialized by the inverse. Alternatively, we can think of the slack bytes being assigned random values. If a client program uses them in a computation that affects its output then it has representation dependence on ddsurfacedesc.

In addition to the statements in Figure 2, an inverse program may consist of the following kinds of statements:

- `(ensr-stmt) ::= ensure((outvar)^+ : (ensr-pred))`
- `(rand-assign) ::= x := *`

where `ensure()` is a call to a constraint solver which solves the constraints that are postponed to the run-time. It finds a satisfying assignment to the list of output variables (of the inverse program) for a given predicate. The run-time values of the input variables (of the inverse program) are substituted in the predicates. We discuss the syntax of (ensr-pred) later on. The assignment $x := *$ assigns an arbitrary value to $x$. It is used when $x$ is unconstrained in the inverse program.
A program $f'$ is a sound inverse of a program $f$ if for every $y$ and for every $z$ that $f'$ returns on $y$ as its input, $f(z) = y$ holds. An inverse program may also consist of assume statements. The presence of non-deterministic choices by ensure() and random assignments may invalidate assumptions during run-time. A program $f'$ is failure-free if no assume statement fails during any of its executions. While our inversion algorithm always synthesizes a failure-free sound inverse, it computes under-approximations of the inverse function of $f$ and hence is an incomplete synthesis procedure.

3. PROGRAM INVERSION

Automatically synthesizing program inverses is a challenging task. In this section, we present an algorithm which works by inverting program statements locally, that is, without modifying the control flow of the input program except for some reordering of statements. While allowing arbitrary statement and control flow changes may seem more powerful, we demonstrate that our algorithm is simple yet effective in inverting a large class of normalization programs, including iterative programs with arrays.

3.1 Loop-free Programs with Scalar Variables

We first illustrate various observations and techniques that we use in inversion of loop-free programs with scalar variables using examples. Consider the following program.

\[
\begin{align*}
1 & \quad y_1 := x_1; \\
2 & \quad y_2 := x_1 + x_2; \\
3 & \quad (\text{Input program}) \\
& \quad x_1 := y_1; \\
& \quad x_2 := y_2 - x_2; \\
& \quad (\text{Inverse program})
\end{align*}
\]

To synthesize an inverse of the input program, we need to compute the input variables $x_1$ and $x_2$ in terms of the output variables $y_1$ and $y_2$. The first statement gives an equality relation $y_1 = x_1$. We treat the output variables as symbolic constants. Thus, $x_1$ is uniquely defined by $y_1$ and we classify $x_1$ as determined. The statement is locally inverted to $x_1 := y_1$ bringing the determined variable on the LHS. This statement is logically equivalent to the original statement and is the first statement of the inverse program. The RHS of the second statement in the input program involves variables $x_1$ and $x_2$. However, $x_1$ is already determined in the prefix of the program. Therefore, $x_2$ is uniquely defined in terms of $y_2$ (an output variable) and $x_1$ (a pre-determined variable) and is also classified as determined. The statement is inverted locally as shown above.

If there are multiple variables on the RHS of an assignment statement that are not determined then one of them can be defined in terms of the others. Consider the following variation of Example 1 with statements reordered.

\[
\begin{align*}
1 & \quad y_1 := x_1 + x_2; \\
2 & \quad y_2 := x_1; \\
3 & \quad (\text{Input program}) \\
& \quad x_1 := y_1; \\
& \quad x_2 := y_2 - x_1; \\
& \quad (\text{Inverse program})
\end{align*}
\]

At the first statement (of the input program), both $x_1$ and $x_2$ are not determined. Say, we define $x_1$ in terms of $y_2$ and $x_2$. However, as $x_2$ is not determined, $x_1$ is also not determined. We instead classify it as a partially determined variable and generate a skip statement in the inverse program. We then eliminate the occurrences of $x_1$ in the suffix of the input program until $x_2$ is determined or we encounter program exit. We discuss the latter case in the sequel. In the second statement, we substitute $y_2 - x_2$ for $x_1$. The resulting statement is inverted locally, since $x_2$ is determined in terms of $y_1$ and $y_2$. All the input variables, namely $x_2$, in the expression $y_2 - x_2$ to which $x_1$ is bound are now determined. Hence, we emit $x_1 := y_2 - x_2$ in the inverse program.

If an input variable $x$ is not known to be a function of other variables at a statement of the input program then the variable is classified as undetermined at the statement.

Let $X$ be the set of input variables of the input program. At every statement, the inversion algorithm computes a partition $(U, D, P)$ of $X$ where $U$, $D$, and $P$ are respectively undetermined, determined, and partially determined variables.

The analysis also computes a binding function $\sigma : X \rightarrow \text{Exp}$ that maps the input variables to program expressions denoted by $\text{Exp}$. If a variable $x$ is partially determined then $\sigma(x)$ is a program expression which evaluates to the same value as $x$. Otherwise, $x$ is bound to a special expression $\bot$. Algorithm 1 inverts program statements locally by computing the partition and the binding function.

Statement Sequences. The function $\text{InvertSequence}$ (Algorithm 2) inverts a given sequence $f$ of program statements. The inversion is achieved through a single forward pass over $f$. Each statement type is treated separately. The tuple $A$ is also refined by the statement specific inversion algorithms.

Assignment Statements. Given an assignment statement $s \equiv y := e$, the function $\text{InvertAssign}$ (Algorithm 3) substitutes all occurrences of the partially determined variables in $e$ by their binding expressions (Line 2). Let $Q = U \cap V(e)$ where $V(e)$ denotes the set of variables occurring in $e$. If
Q = ∅ then all variables in e are already determined and assigned in the inverse of the prefix of the program ending at the current statement. Nevertheless, their values should conform with the equality y = e. This constraint is embedded in the inverse program as an assert statement.

If only one variable of e, say x, is undetermined then s is inverted to x := e′. Given an assignment statement y := e and a variable x appearing in e, LocalInvert returns an expression e′ such that x = e′ is logically equivalent to y = e. LocalInvert works by algebraic rewriting of an assignment such as y := a * x + b to x := (y - b)/a assuming that every operator (such as + and *) has an inverse (such as − and /). The variable x is removed from U and added to D by the function MarkDEmitD (Algorithm 4). Given a set W ⊆ U of variables, MarkDEmitD marks them as determined. If all variables occurring in the binding expression σ(x) of a partially determined variable x are now determined then x is also determined (Line 5). The function MarkDEmitD emits the assignment statement x := σ(x). This process is repeated until no partially determined variable can be marked as determined. The bindings carried by σ, discharged as assignment statements by MarkDEmitD, cause statement reordering as discussed Example 2.

If more than one undetermined variables occur in e then one of them, say x, is marked as partially determined and bound to the expression LocalInvert(s,x) (Lines 11-12).

Assume Statements. Given an assume statement assume(p), the function InvertAssume (Algorithm 5) substitutes all occurrences of partially determined variables in p by their binding expressions. All variables in U ∩ V(p) are marked as determined. As explained later, we ensure that these variables take values that conform to p at the current statement. Some variable bindings may be discharged by MarkDEmitD. Finally, the statement assume(p) is generated.

Conditional Statements. Given a conditional statement s ≡ if (p) then f₁ else f₂ end, the function InvertConditional (Algorithm 6) processes p similar to the predicate of an assume statement and inverts the branches f₁ and f₂ individually. Let A₁ = (U₁, D₁, P₁, σ₁), i ∈ {1, 2}. If P₁ = P₂ and σ₁ = σ₂ then Join(A₁, A₂) = (U₁ ∪ U₂, D₁ ∪ D₂, P₁, σ₁).

A variable x ∈ D \ D₁ = U₁ ∩ D₂ is marked as determined by the join operation but is undetermined in f₁. To force the determined status, a random assignment x := s is appended to the inverse of f₁ (Line 8). Similarly, for f₂. This ensures that the future uses of x can be resolved to be determined, unconditionally, enabling local inversion. The conditional statement for the inverse is then formed by composing the inverses of the branches. If P₁ ≠ P₂ or σ₁ ≠ σ₂ then one of the branches is eliminated1 (Lines 14-17).

In the following program, the variables x₂ and x₃ are determined in the if-branch and the else-branch respectively.

We insert random assignments x₃ := * and x₂ := * in the if-branch and the else-branch respectively to force both x₂ and x₃ to be determined along both the branches. We discuss the predicate ϕ(x₁, x₂) later.

Program Exit. Once f is processed completely, the function EmitRemaining appends random assignments to the undetermined variables used in the binding expressions to f′. For each x ∈ P, x := σ(x) is then emitted.

Ensuring Failure-free Execution. The inversion algorithm may generate random assignments to variables while processing conditional statements (Algorithm 6, Lines 8-9). Further, new assume statements may be added to the inverse program if all variables on the RHS of an assignment statement are already determined (Algorithm 3, Line 4).

Consider Example 3 again. If x₂ gets assigned a value ≠ y₃ in the else-branch then the assume statement in the inverse program fails. To avoid such failures, the function EmitEnsure (Algorithm 1, Line 10) performs the following operations: (1) it computes the weakest precondition wp of assume statements, (2) emits an ensure statement at the program entry to satisfy the precondition, and (3) removes random assignments that may violate assume statements.

Given the antecedent Hoare triple of a wp rule in the inverse program f′, EmitEnsure generates the consequent triple to obtain a transformed version of the inverse program.

1It is possible to combine inequal sets of partially determined variables (e.g., by unifying their binding expressions) but, for simplicity, we restrict it to the present form.
Algorithm 5: Inversion of Assume Statements

Routine: InvertAssume(A, s)
1 Let s = \(\omega\) or \(\omega_0\)
2 for each \(x \in D \rightarrow p \rightarrow \text{subst}(p, x, \sigma(x))\)
3 \((A, s') \leftarrow \text{MarkDelete}(A, U \cup V(p))\)
4 return \((A, \text{Append}(s', s))\)

Algorithm 6: Inversion of Conditional Statements

Routine: InvertConditional(A, s)
1 Let \(s = \omega \) or \(\omega_0\)
2 for each \(x \in D \rightarrow p \rightarrow \text{subst}(p, x, \sigma(x))\)
3 \((A, s') \leftarrow \text{MarkDelete}(A, U \cup V(p))\)
4 \((A_1, f_1) \leftarrow \text{InvertSelect}(A, f_1)\)
5 \((A_2, f_2) \leftarrow \text{InvertSelect}(A, f_2)\)
6 if \(P_1 = P_2 \land \sigma_1 = \sigma_2\) then
7 \(A \leftarrow \text{Join}(A_1, A_2)\)
8 for each \(x \in D \setminus D_0\) do \(f'_2 \leftarrow \text{Append}(f'_2, x := 3)\)
9 for each \(x \in D \setminus D_0\) do \(f'_1 \leftarrow \text{Append}(f'_1, x := 3)\)
10 \(s' \leftarrow \text{Append}(s', f'_1)\) if \(p\) then \(f'_1\) else \(f'_2\) end
11 \((A, s'') \leftarrow \text{MarkDelete}(A, \emptyset)\)
12 \(s'' \leftarrow \text{Append}(s', s'')\)
13 else
14 \(A \leftarrow \text{Select}(A_1, A_2)\)
15 \(s'' \leftarrow \text{Append}(s', f'_1)\) else assume(false) end
16 \(s'' \leftarrow \text{Append}(s', (f'_1)\) if \(p\) then \(f'_1\) else assume(false) end
17 \(s'' \leftarrow \text{Append}(s', (f'_1)\) if \(p\) then assume(false) else \(f'_2\) end
18 end
19 return \((A, s'')\)

The rule involving \(x := */\) and a postcondition \(\varphi\) where \(x \in V(\varphi)\) eliminates the random assignment to \(x\) as it may violate the postcondition \(\varphi\). Instead, \(x\) is assigned a value that satisfies \(\varphi\) by the ensure() statement at the program entry. The assignment \(x_2 := */\) in Example 3 is eliminated by this. The last rule emits the ensure statement. The predicate \(\varphi(x_1, x_2)\) in Example 3 is generated using these rules: \(\varphi(x_1, x_2) = (x_1 > 0 \land y_2 = y_1) \lor (x_1 \leq 0 \land y_1 = x_2)\). The ensure() statement is a call to a constraint solver to find satisfying assignments to \(x_1\) and \(x_2\) subject to the predicate \(\varphi(x_1, x_2)\) where the run-time values of \(y_1\) and \(y_2\) are substituted. We discuss the constraint solver in Section 4.

Correctness. Each statement of the input program is converted to a logically equivalent statement in synthesis of an inverse while maintaining the control flow of the inverse program. Given values of the output variables \(y\), an input variable \(x\) is assigned exactly once in the inverse program. This guarantees that the value of \(x\) conforms with all its prior relations with \(y\). A subsequent statement \(s\) of the input program involving \(x\) is also mapped to a logically equivalent statement \(s'\) in the inverse program. It can be either an assignment to another variable \(x'\) or an assume statement. Further, the constraint to the ensure() statement guarantees that no assume statement fails during run-time. The inverse however many not be complete, in a sense, that given \(y\) it may not produce all \(x\)’s such that \(y = f(x)\). This is because of handling of conditionals when partially determined variables along the two branches do not match and limitations of the constraint solver used in ensure() statements.

3.2 Iterative Programs with Arrays

We now extend the inversion algorithm to iterative programs with arrays. We use the following adaptation of Example 1 to illustrate the basic ideas behind loop inversion.

1 for \(i = 1\) to \(10\) do
2 \(y_1[i] := x_1[i];\)
3 \(y_2[i] := x_1[i] + x_2[i];\)
4 end

(Example 4)

Algorithm 7 performs inversion of a loop \(s\). It copies the loop-structure \(\ell'\) of the loop \(s\) to the inverse program (Lines 1, 9) but inverts the statements within the loop body locally as we did for loop-free programs (Line 5). We now describe the extensions required to do this. In the sequel, the term reference denotes either a scalar variable such as \(i\) or an indexed array variable such as \(x_1[i]\).

Consider an assignment statement \(y_1[e_0] := x_1[e_1] + x_2[e_2]\). A reference such as \(x_1[e_1]\) or \(x_2[e_2]\) is a top-level reference. These are given by the function TopLevelRefs. We can rewrite the assignment statement to express any top-level reference as a function of other top-level references and the LHS, due to the invertibility assumption (Section 2). The top-level references play a role analogous to (scalar) variables in our original algorithm. A reference occurring inside an array index expression (such as \(e_0, e_1, e_2\)) is a nested reference. These are given by the function NestedRefs. They are treated differently. In particular, they must be determined before we can process this assignment statement. Hence, we (conservatively) mark the nested references as being determined.

Symbolic Partition Representation. We extend our scheme for partitioning a statically fixed number of (scalar) variables into \((U, D, P)\) to similarly partition a statically unbounded number of (array) locations. The current iteration of a simple loop is represented symbolically by its counter \(i\). The status of locations that become determined or partially determined in the current iteration is maintained just as in the case of loop-free code. In Example 4, the set of locations determined in the current iteration after the first and second statement are \(\{x_1[i]\}\) and \(\{x_1[i], x_2[i]\}\) respectively.

We utilize a symbolic representation to describe the set \(D\) of array locations that were determined in iterations of the loop preceding the current iteration (as a function of the iteration index). In the above example, the symbolic representation of \(D\) takes the form \(\{x_1[i'], x_2[i']\} | 1 \leq i' < 10 \land i' < i\), which describes the set of determined locations at the beginning of iteration \(i\). In general, the symbolic representation is of the form \(\{S | Q\}\) where \(S\) is a set of symbolic input array locations with index expressions over some free variables \(e.g., i, Q\) is a constraint over these free variables. \(Q\) is of the form \(B_i \land L_i\), where \(B_i\) is the predicate \((1 \leq i' \leq n_i)\) with \(n_i\) as the upper bound of \(i\), and \(L_i\) is the predicate \(i' < i\). This can be extended to nested loops using the lexicographical order over iteration vectors.
Algorithm 7: Inversion of Loops

Routine: InvertLoop(A, s)
1 $b' \leftarrow$ GetBody(s)
2 $U \leftarrow U \cup \text{TopLevelRefs}(s) \cup \text{NestedRefs}(s)$
3 $D \leftarrow D \cup \{S \mid Q\}$
4 $(A, b') \leftarrow \text{InvertSequence}(A, b)$
5 $b' \leftarrow \text{.emitPRemaining}(A, b')$
6 $s' \leftarrow \text{NonAliasConstr}(b') \land \text{LoopBoundConstr}(b')$
7 $s' \leftarrow \text{assume}(s')$
8 $s' \leftarrow \text{Append}(s', \text{InsertBody}(b', b'))$
9 return $(A, s')$

$S$ is a list of all array references that become determined in a particular iteration $i'$. To simplify construction of $S$, we conservatively treat every location referenced in a given iteration as determined at the end of that iteration. This reflects in the initialization of the set $D$ in Line 4. Consequently, we do not carry partially determined variables across loop iterations. We therefore emit assignments to partially determined variables at the end of the loop body (Line 6).

Generalized Assignment Inversion. Algorithm 8 shows how an assignment statement is processed during inversion of a loop body. If it is an assignment to a local variable, say $k$, then the RHS $e_k$ has only $k$ as a variable (Section 2). We simply copy the statement to the inverse program (Line 1).

Otherwise, we do substitutions for partially determined references in $e$, as before, but substitutions are restricted to top-level references (Line 4). Next, we mark nested references as determined (Lines 5-7). Assume that we want to invert $y[e_0] := x[e_1]$ into $x[e_1] := y[e_0]$. Clearly, we want every nested reference $r$ occurring in $e_0$ and $e_1$ to be assigned a value before this inverted statement executes in the inverse program. Marking $r$ as determined at this point captures this requirement. They are assigned values by the ensure() statement by solving constraints generated by the inversion algorithm over them as discussed shortly.

We then identify the top-level references that are undetermined (Lines 8-15). Unlike in the earlier algorithm, we may not be able to precisely categorize the status of a reference $r$. In this case, we determine the condition $\psi$ under which $r$ is undetermined (as explained soon) and accumulate the set of such constraints and treat $r$ as being undetermined.

We identify the status of a reference $r$, by checking whether $r$ became determined either earlier in the current iteration or in some previous iteration. The latter condition is checked as $r \in \{S \mid Q\}$. We treat $r$ as undetermined only if we can establish that $r$’s status is undetermined in every iteration. Otherwise, we treat it as being determined. This computes a loop-invariant partition of the top-level references and enables us to invert the loop body locally (Lines 16-18).

Consider the following example.

\begin{verbatim}
1 for i := 1 to 10 do
2 y[i] := x[i] + x[i + 3];
3 end
\end{verbatim}

(Example 5)

The set of locations determined in iterations preceding iteration $i$ is given by $\{S \mid Q\}$, where $Q = (1 \leq i' \leq 10) \land (i' < i)$, and $S = \{x[i'], x[i' + 3]\}$. Now, consider the statement $y[i] := x[i] + x[i + 3]$. Intuitively, $x[i]$ has already been referenced in iteration $i - 3$ using the reference $x[i + 3]$ (for $i > 3$). More precisely, since $x[i] \in \{S \mid Q\}$, $x[i]$ is marked as determined. Since $x[i + 3] \notin \{S \mid Q\}$, $x[i + 3]$ is marked as undetermined. The statement is then inverted as shown.

The function $\text{CheckIfUndetermined}$ decides the status of a reference unambiguously as “true” or “false” for affine array index expressions. Given a reference $x[e]$, it checks whether $x[e]$ is determined in the current iteration or $x[e] \in \{S \mid Q\}$ which amounts to $3^{e'} \cdot i \cdot (1 \leq i \leq n_i) \land Q \land (\forall e'[e] \in S \ e = e')$ and is solved as an integer linear program.

In inversion of iterative programs with arrays, we, by default, initialize array locations. We use $x := *$ as a shorthand for assignment of random values to all locations of array $x$. In Example 5, $x[1], x[2]$, and $x[3]$ take random values assigned to them by $x := *$ whereas $x[4], \ldots, x[13]$ are assigned to by the inverse program subsequently. Similar to the case of loop-free code, the wp computation ensures failure-free execution in presence of random assignments.

Exploiting Dynamic Constraint Solving. We now illustrate how we handle references with an ambiguous status by collecting constraints that are solved at run-time when the inverse program executes. Consider the following example.

\begin{verbatim}
1 ensure(x[1]; \forall i. \varphi(i));
2 for i := 1 to 8 do
3 y[i] := x[2][x[1][i]];
4 end
\end{verbatim}

(Example 6)

The set $D$ of determined locations of $x_2$ at the beginning of iteration $i$ is given by $\{x_2[x_1[i']] \mid (1 \leq i' \leq 8) \land (i' < i)\}$. When processing the assignment $y[i] := x_2[x_1[i]]$, in Line 10, the algorithm tries to determine whether location $x_2[x_1[i]]$ is in the set of locations $D$. Unlike earlier examples, the answer in this case depends on the values assigned to $x_1$. The function $\text{CheckIfUndetermined}$ symbolically simplifies negation of the above membership test and returns the condition $\psi(i) = \forall i' \cdot ((1 \leq i' \leq 8) \land (i' < i) \Rightarrow (x_1[i'] \neq x_1[i])$. The function $\text{AddConstraint}$ in Line 13 collects the set of all such constraints $\psi(i)$ generated during the processing of
the loop body for each array reference separately. Once we finish processing the loop body, we lift the constraint \( \psi(i) \) (on a single iteration \( i \)) into a constraint on all iterations by quantifying over all possible values of \( i \). In the example above, we have \( \forall i \cdot \psi(i) = \forall i \cdot (1 \leq i \leq 8) \Rightarrow \psi(i) \). We call these constraints non-aliasing constraints. The function NonAliasConst inserts them as an assume statement preceding the loop (Algorithm 7, Line 7). This allows us to consider the reference \( x_2[x_1[i]] \) as undetermined, and invert the assignment statement. The constraint is then propagated backwards to form a part of the ensure() statement.

The non-aliasing constraints are universally quantified constraints involving integer arithmetic and arrays due to indirection in array indexing similar to the example above. Giving a formal characterization of such constraints is beyond the scope of this paper. We however introduce them here since we encounter them in the Tiff case study. In Section 4, we discuss our specific implementation to solve them.

**Loop-bound Constraints.** Our algorithm inverts the loop body locally, keeping the loop-structure the same. However, the loop bounds may be input (resp. output) variables of the input (resp. inverse) program. An inverse is sound if it accesses exactly those locations of an output array \( y \) that are assigned to by the input program. To ensure this, we require the user to specify the size of an output array. The specification is a dependent type signature as the array size depends on values of the input variables used as bounds of the loops indexing over the array. We provide a function \( \text{sum}(p,q) \) to indicate array sizes. \( \text{sum}(p,q) \) is summation of \( p, q \) times. The arguments to \( \text{sum} \) can be input variables including array subscripts. Consider the following program.

```plaintext
Input: int n, m, x[\text{sum}(n, m)];
Output: int n', m', y[\text{sum}(n, m)];
1 k := 1;
2 for i := 1 to n do
3 for j := 1 to m do
4 y[k] = x[m * (i - 1) + j]; k := k + 1;
5 end
6 end
7 n' := n; m' := m; /* instrumentation */
```

Consider the variable \( n \) that appears in the dependent type signature. Its run-time value is captured as an additional output variable \( n' \) by instrumenting the input program. Since we are interested in equality of array sizes, we emit what we call as loop-bound constraints in the inverse program (Algorithm 7, Line 7). The loop-bound constraint for the above program is \( \text{sum}(n, m) = \text{sum}(n', m') \). Note that \( n \) and \( m \) are the variables here, whereas \( n' \) and \( m' \) take the values generated by the input program. In inversion, the instrumentation code in the input program is ignored and these constraints are solved by the ensure() statement.

**Correctness.** The algorithm ensures that the control flow of the inverse program \( f' \) is isomorphic to that of the input program \( f \). Further, the conditional statements, loop-structures, and assignments to local variables match syntactically. Let \( x[n] \) be the output of \( f' \) on input \( y'[m] \). If \( f' \) takes a path \( p \) in the execution then \( f \) when executed on \( x[n] \) also takes the same path. Let \( y[m] \) be the output of \( f \) on \( x[n] \). The loop-bound constraint guarantees that \( m' = m \).

We restrict the discussion to array locations only and argue that \( y[k] = y'[k] \), for all \( k \), such that \( 1 \leq k \leq m \). For simplicity, let there be only one syntactic assignment to \( y \) in \( f \) of the form \( y[e_1] := x[e_2] \) and the corresponding inverse statement be \( x[e_2] := y[e_1] \). Let \( e_1 \) be function of a local variable \( h \) and \( e_2 \) be function of a loop counter \( i \). Due to the control flow matching described above, whenever \( i \) takes the same value in both the programs, the corresponding instances of \( h \) also take the same value. This implies that whenever \( e_2 \) evaluates to a value, say \( k' \), in either program, \( e_1 \) evaluates to the same value, say \( k \), in both the programs. We have assumed that \( f \) assigns to an array location \( y[k] \) exactly once (Section 2) and the inversion algorithm guarantees that \( x[k'] \) is assigned exactly once. Thus, \( x[k'] = y'[k] \) and \( y[k] = x[k'] \). Further, as discussed in Section 3.1, the ensure() statement guarantees failure-free execution.

**Theorem 1.** A program \( f' \) synthesized by Algorithm 1 is a sound inverse of the input program \( f \) and is failure-free.

### 4. EXPERIMENTAL RESULTS

We demonstrate effectiveness of our testing methodology (Figure 1) on five popular Tiff image [5] clients. One can find a large number of online discussions (e.g., [1, 2, 3]) about a client’s dependence on specific encoding styles used in common Tiff producers. Due to these reasons, Tiff which stands for “Tagged Image File Format” is also jokingly called “Thousands of Incompatible File Formats” [27], but in fact, the problem is representation dependence of clients. The specification clearly states: “Where there are options, Tiff writers can use whichever they want. Baseline Tiff readers must be able to handle all of them.” (Section 7, [5]).

#### 4.1 Normalization of Tiff Images

We restrict our attention to strip-based grayscale images without compression. We cover the flexibility in the choice of orientation, rows per strip, and layout of strips. We believe that several other variations (e.g., tile-based, colored, etc.) and equivalences (e.g., byte and fill ordering, planar configuration, etc.) can be analyzed with our technique. However, compression seems beyond the scope of our current algorithm. Without loss of generality, we take the following definition of the Tiff data structure. We use the LibTiff library [21] to read and write a Tiff file.

```plaintext
typedef struct _tiff {
  int orientation, n, w, r;
  int x_1[N], x_2[N], x_3[N][N];
} tiff;
```

(a) Tiff data structure  (b) Canonical representation

The canonical representation of a tiff data consists of a 2D bitmap \( y[N][N] \) where \( N \) is the array size. As shown in Figure 3(a), \( x_1[i] \) gives the offset of the \( i \)th strip, \( x_2[i] \) gives
We solve these (quantified) constraints through heuristic instantiation. The relation \( \prec \) is the lexicographic ordering. We instantiate all non-constant quantifier bounds and a sufficient number of variables in a non-linear constraint until it becomes linear. In particular, we instantiate \( n \) by considering three cases: \( n < n' \), \( n = n' \), and \( n > n' \). The non-aliasing inequality constraint is strengthened to the following strict ordering constraints:

\[
\forall i \cdot (x_1[i] + x_2[i] \leq x_1[i+1]) \lor \forall i \cdot (x_1[i+1] + x_2[i+1] \leq x_1[i])
\]

where \( 1 \leq i < n \). After instantiating \( n \) to, say \( k \), the loop-bound constraint \( \sum(n, x_2[i]) = \sum(n', x_2[i]) \) is expanded to \( x_2[1] + \ldots + x_2[k] = x_2[1] + \ldots + x_2[n'] \). Recall that \( x_2 \) and \( n' \) have pre-defined values at run-time. Similarly, the universal quantification is compiled into a finite conjunction with the array subscript \( i \) (the quantified variable) ranging from 1 to \( k - 1 \). The resulting constraint is solved using Yices [16]. Typically, an SMT solver, including Yices, returns the same solution every time for a constraint. To generate multiple test cases, we solve the constraints multiple times by incrementally adding a constraint to avoid the previous satisfying assignment. One of the satisfying assignments is then picked at random. The inverse program then executes deterministically and copies the pixel data appropriately.

We take a TIFF file displayed correctly by all the clients and generate equivalent TIFF images automatically. Despite the heuristic as well as logical simplification of the constraints and incompleteness in inversion and constraint solving, we were able to find several representation dependence bugs. Figure 5 aggregates the results along various undetermined fields of TIFF for a test run involving 40 images generated from a single image. Layout indicates whether the logically adjacent strips can be physically rearranged. A \( \checkmark \) indicates that the application passed all test cases and \( \times \) means it failed for at least one. The success of a test is determined by whether the client is able to display the image correctly. Picasa 3.6 fails for combinations of orientation and layout changes and without code inspection, we cannot isolate the problem to one of the fields. We therefore conservatively mark its dependence on both.

The automated inversion offers several advantages over manually written equivalence-preserving transformations:

- While an (equivalence-preserving) transformation for changing orientation is easy to write manually, other variations require constraint solving. Our inversion algorithm derives the desired constraints automatically.
- Intuitively, to produce equivalent TIFF images, it is necessary to implicitly go through the canonical representation. For instance, to redistribute the strips, one has to first construct the bitmap. Thus, the notions of canonical representation and normalization program are arguably central to this kind of testing.
- Since we can generate a large number of tests automatically, the “no representation dependence” conclusions (\( \checkmark \) above) are more plausible.

**Figure 5: Test Results for TIFF Clients**

<table>
<thead>
<tr>
<th>Application</th>
<th>Orientation</th>
<th>Rows per strip</th>
<th>Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>FastStone 3.6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>KView 3.5.4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GIMP 2.2.13</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Open Office 2.0.4</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Picasa 3.6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

The number of rows in the \( i \)-th strip, and \( x_3[x_1[i]+j][k] \) gives the pixel at the \( j \)-th row and \( k \)-th column of the \( i \)-th strip.

The actual number of strips is \( n \). Generating the canonical representation involves reading strips from 1 to \( n \). The orientation flag indicates whether the pixel data is stored from left-to-right or right-to-left and top-to-bottom or bottom-to-top.

The actual number of strips is \( n \). Generating the canonical representation involves reading strips from 1 to \( n \). The orientation flag indicates whether the pixel data is stored from left-to-right or right-to-left and top-to-bottom or bottom-to-top. Figures 3(b)-(c) indicate canonical representation involves reading strips from 1 to \( n \). Generating the canonical representation involves reading strips from 1 to \( n \). The orientation flag indicates whether the pixel data is stored from left-to-right or right-to-left and top-to-bottom or bottom-to-top.

- The orientation flag indicates whether the pixel data is stored from left-to-right or right-to-left and top-to-bottom or bottom-to-top. Figures 3(b)-(c) indicate canonical representation involves reading strips from 1 to \( n \). Generating the canonical representation involves reading strips from 1 to \( n \). The orientation flag indicates whether the pixel data is stored from left-to-right or right-to-left and top-to-bottom or bottom-to-top.

The constraint (2) corresponds to the assume statements in the input program. The constraint (3)–(4) is the non-aliasing constraint. The relation \( \prec \) is the lexicographic ordering. We solve these (quantified) constraints through heuristic instantiation (cf. [14, 17]). We instantiate all non-constant quantifier bounds and a sufficient number of variables in a non-linear constraint until it becomes linear. In particular, we instantiate \( n \) by considering three cases: \( n < n' \), \( n = n' \), and \( n > n' \). The non-aliasing inequality constraint is strengthened to the following strict ordering constraints:

\[
\forall i \cdot (x_1[i] + x_2[i] \leq x_1[i+1]) \lor \forall i \cdot (x_1[i+1] + x_2[i+1] \leq x_1[i])
\]

where \( 1 \leq i < n \). After instantiating \( n \) to, say \( k \), the loop-bound constraint \( \sum(n, x_2[i]) = \sum(n', x_2[i]) \) is expanded to \( x_2[1] + \ldots + x_2[k] = x_2[1] + \ldots + x_2[n'] \). Recall that \( x_2 \) and \( n' \) have pre-defined values at run-time. Similarly, the universal quantification is compiled into a finite conjunction with the array subscript \( i \) (the quantified variable) ranging from 1 to \( k - 1 \). The resulting constraint is solved using Yices [16]. Typically, an SMT solver, including Yices, returns the same solution every time for a constraint. To generate multiple test cases, we solve the constraints multiple times by incrementally adding a constraint to avoid the previous satisfying assignment. One of the satisfying assignments is then picked at random. The inverse program then executes deterministically and copies the pixel data appropriately.
5. RELATED WORK

For an abstract datatype, representation exposure is said
to occur when an internal implementation of an aggregate
object is accessible for modification outside the aggregate [12,
20]. Ownership type systems [13, 9, 6], and types for reference
immutability [7] have been designed to prevent this.
But these systems still allow the inner workings of the ab-
stract datatype to be seen by its users. This phenomenon is
called observational exposure [10]. Our work can be viewed
as checking both kinds of exposures using testing. We check
if a client makes stronger logical assumptions on the rep-
resentation of data than those allowed by the specification.
We are unaware of any type system that can perform the
equivalent of what we are doing.

Grammar-based testing [22, 26] is used for testing func-
tionality of grammar-driven software by enumerating test
data from a grammar definition. In our work, the explo-
rution of the syntactic space of data is driven by data equiv-
ance specification in terms of normalization programs.
Ko-
rat [8] systematically enumerates data instances from a data
structure invariant specified as a Java predicate. Test case
generation for representation dependence is a technically dif-
ferent problem. Given a test suite, metamorphic testing [11,
18, 23] generates new test cases to test a program using
known relations, called metamorphic relations, between in-
puts and outputs of the program. In contrast, our test gen-
eration is based on algorithmic inversion of normalization
programs with a more specific testing objective.

Dijkstra [15] observed that program inversion can be seen
as running a program backwards to map output to input. 
Ross [24] encodes relational semantics of programs and uses
logic programming to compute inverses. Our approach uses
a forward reasoning and inverts program statements locally.
We can effectively deal with imperative programs involving
conditionals, loops, and arrays. Relational calculus has
been used to derive program inverses, mainly of functional
programs [19, 25]. The derivations are typically deductive
whereas we give an algorithmic solution for inversion.

6. CONCLUSIONS

Representation dependence is a frequent problem in prac-
tice. Given a normalization program as a mapping from
a data instance to its canonical representation, we invert
it algorithmically to automate generation of test cases for
detecting representation dependence. We have tested TIFF
clients effectively with this approach.

The technique of local inversion and deferring of statically
unsolvable constraints to run-time make the difficult prob-
lem of program inversion practically solvable for an interesting
class of programs including a class of iterative programs
with arrays. However, our approach has several limitations,
namely, imprecise join operation and syntactic restrictions
on input programs outlined in Section 2. Program inversion
thus still remains an open problem at large. Automated
testing and analysis for representation dependence appears
to be a challenging yet fruitful task ahead.

7. REFERENCES

may/000730.html.
and language for reference immutability. In OOPSLA,
automated testing based on java predicates. In ISSTA,
[10] J. Boyland. Why we should not add readonly to Java
Metamorphic testing: a new approach for generating
next test cases. Technical report, Hong Kong
University of Science and Technology, 1998.
representation exposure. In Proc. of the Work. on
Object-Oriented Tech., pages 149–151, 1999.
theorem prover for program checking. J. ACM,
quantified verification conditions using satisfiability
of a binary tree and its inversion. Formal Development
[20] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and
R. Holt. The Geneva convention on the treatment of
[22] P. M. Maurer. Generating test data with enhanced
system testing of programs without test oracles. In ISSTA,
inversion of imperative computation. Formal Asp.
[25] B. Schoenmakers. Inorder traversal of a binary heap
and its inversion in optimal time and space. In MPC,