11-2010

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

Recommended Citation

Aditya Kanade, Rajeev Alur, Sriram Rajamani, and Ganesan Ramalingam, "Representation Dependence Testing Using Program Inversion", *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)* , 277-286. November 2010. http://dx.doi.org/10.1145/1882291.1882332

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 oneto- 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

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.

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

developer (of the data definition) specifies the equivalence relation on data representations by providing a *normalization* program (the semantics function) f that maps representations to the values they denote. In particular, $f(x)$ can be viewed as the canonical representation of x. Finding alternative representations of x essentially requires identifying other representations z such that $f(z) = f(x)$.

Program Inversion. Representation fuzzing is essentially a constraint solving problem: given f and x we want to determine z such that $f(z) = f(x)$. Our approach is to decompose this problem into two parts. First, given f, we automatically synthesize a program f' that represents the inverse of f. The inverse will typically be a one-to-many function realized via non-determinism. For every y and for every z that f' returns on y as its input, $f(z) = y$ must hold. This is the correctness criterion for the inversion problem. Second, given x, we repeatedly apply f' to $f(x)$ to produce different z that are logically equivalent to x.

This staged approach lets us do expensive (symbolic) constraint solving once (offline) during the program inversion phase, when possible. However, some of the constraint solving may be hard if we treat $y = f(x)$ symbolically as an unknown but may be easier if the value of y is known. In such cases, we embed these constraints in the synthesized inverse program so that they are solved dynamically (when f' is applied to a given value of y).

The key technical challenge we address in the paper is that of synthesizing the inverse function. In the simple case of straight-line code computing outputs $(y_1, \cdots, y_m)$ from inputs $(x_1, \cdots, x_n)$, we generate the inversion code by symbolically solving a collection of equations for $(x_1, \cdots, x_n)$. In the general case, conditionals, loops, and arrays complicate inversion. One of the essential ideas we exploit for inversion in the general case is to synthesize an inverse program that has a control-flow structure isomorphic to the original program, using local inversion (via symbolic equation solving) to invert loop-free code fragments.

TIFF Case Study. TIFF is a widely used image format that permits multiple representations of the same image and is known to raise compatibility issues among programs. For instance, a TIFF image consists of an *orientation* flag which determines whether the scanlines of the image bitmap are stored from left-to-right or right-to-left and top-to-bottom or bottom-to-top. Thus, there are multiple equivalent representations of a TIFF image. We use TIFF as a case study for representation dependence testing.

Figure 1 summarizes our methodology for testing a client program C for representation dependence with TIFF images as an example. The normalization program f converts a TIFF representation to a canonical bitmap representation. Our algorithm automatically synthesizes f', the inverse of f. We then take a TIFF file TIFF1 and compute a bitmap BM $= f($TIFF1$)$. Applying f' to BM produces an alternative TIFF representation TIFF2 of the same image as the bitmap BM. We run C on both TIFF1 and TIFF2 and report any difference in the behavior (or output) of C as a data representation dependence. The non-determinism in f' (*e.g.*, the choice of *orientation*) means that different runs of f' can produce different outputs. Thus, repeating the above procedure multiple times increases the effectiveness of testing even when we use the same input TIFF1.

The normalization program f for TIFF consists of several programming constructs like conditionals, nested loops, unbounded arrays, and indirection 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:

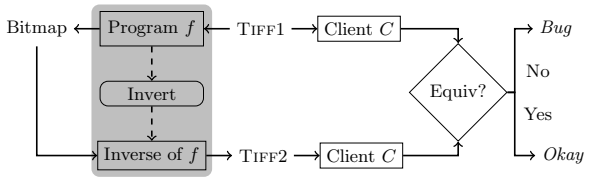

Figure 1: Automated Testing for Representation Dependence (of Clients of TIFF Images)

- 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.

$$
\begin{array}{lcl}
\langle \text{program} \rangle & ::= & \langle \text{stmt} \rangle^+ \\
\langle \text{stmt} \rangle & ::= & \langle \text{smpl-stmt} \rangle \mid \langle \text{cond-stmt} \rangle \mid \langle \text{loop-stmt} \rangle \\
\langle \text{smpl-stmt} \rangle & ::= & \langle \text{outvar} \rangle := \langle \text{exp} \rangle \mid \text{assume}(\langle \text{pred} \rangle) \\
\langle \text{cond-stmt} \rangle & ::= & \textbf{if } (\langle \text{pred} \rangle) \textbf{ then } \langle \text{stmt} \rangle + \textbf{ else } \langle \text{stmt} \rangle + \textbf{ end} \\
\langle \text{loop-stmt} \rangle & ::= & \textbf{for } \text{i} := 1 \text{ to n } \textbf{do } \langle \text{body} \rangle \textbf{ end} \\
\langle \text{body} \rangle & ::= & \langle \text{smpl-stmt} \rangle^+ \mid \langle \text{loop-stmt} \rangle \\
\langle \text{invar} \rangle & ::= & x \mid x_1 \mid \ldots \mid x_2[\langle \text{ind-exp} \rangle]^+ \mid \ldots \mid m \mid n \\
\langle \text{outvar} \rangle & ::= & y \mid y_1 \mid \ldots \mid y_2[\langle \text{ind-exp} \rangle]^+ \mid \ldots \\
\langle \text{indvar} \rangle & ::= & i \mid j \mid k \\
\langle \text{ind-exp} \rangle & ::= & \langle \text{exp} \rangle \mid \langle \text{scalar-invar} \rangle * \langle \text{exp} \rangle \mid x[\langle \text{exp} \rangle] \\
 & \mid & \pm \langle \text{ind-exp} \rangle \pm \langle \text{ind-exp} \rangle
\end{array}
$$

Figure 2: The Syntax of Normalization Programs

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$[d.\text{pitch} * i + j] = d'$.surface$[d'.\text{pitch} * i + j]$ 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.

```
   /* 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 $\langle \text{exp} \rangle$ and predicates $\langle \text{pred} \rangle$ 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 $y[\ell]$. 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 y 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.

```
   /* Input:n, Output:d */
1  ensure(d.pitch :  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:

$$
\begin{array}{lcl}
\langle \text{ensr-stmt} \rangle & ::= & \text{ensure}(\langle \text{outvar} \rangle^+ : \langle \text{ensr-pred} \rangle) \\
\langle \text{rand-assgn} \rangle & ::= & x := *
\end{array}
$$

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 $\langle \text{ensr-pred} \rangle$ later on. The assignment $x := *$ assigns an arbitrary value to x. It is used when x is unconstrained in the inverse program.

Algorithm 1: Inversion of Programs

Routine: Invert(f)
Input: A program f in the language of Figure 2
Output: A program f' that is semantic inverse of f
1 $U \leftarrow X$; $D \leftarrow \emptyset$; $P \leftarrow \emptyset$; $\sigma = \{x \mapsto \perp | \ x \in X\}$
2 Let A denote (U, D, P, σ)
3 $(A, f') \leftarrow$ InvertSequence(A, f)
4 $f' \leftarrow$ EmitPRemaining(A, f')
5 **return** EmitEnsure(f')

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.

1 $y_1 := x_1$;	**1** $x_1 := y_1$;
2 $y_2 := x_1 + x_2$;	**2** $x_2 := y_2 - x_1$;
(Input program)	(Inverse program)

(Example 1)

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.

Algorithm 2: Inversion of Statement Sequences

Routine: InvertSequence(A, f)
Input: $A = (U, D, P, \sigma)$ and a statement sequence f
Output: Updated tuple A and inverse f' of f
1 $f' \leftarrow [\,]$
2 **while** $f \neq [\,]$ **do**
3 $s \leftarrow$ Car(f)
4 **switch** s **do**
5 **case** $y := e$ $(A, s') \leftarrow$ InvertAssign(A, s)
6 **case** assume(p) $(A, s') \leftarrow$ InvertAssume(A, s)
7 **case if** (p) **then** s_1 **else** s_2 **end**
8 $(A, s') \leftarrow$ InvertConditional(A, s)
9 **case for** $i := 1$ *to* n **do** *body* **end**
10 $(A, s') \leftarrow$ InvertLoop(A, s)
11 **end**
12 $f' \leftarrow$ Append(f', s')
13 $f \leftarrow$ Cdr(f)
14 **end**
15 **return** (A, f')

1 $y_2 := x_1 + x_2$;	**1** skip;
2 $y_1 := x_1$;	**2** $x_2 := y_2 - y_1$;
3 skip;	**3** $x_1 := y_2 - x_2$;

(Example 2)

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* $\sigma : X \rightarrow Exp$ that maps the input variables to program expressions denoted by 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 $\perp$. Algorithm 1 inverts program statements locally by computing the partition and the binding function.

Statement Sequences. The function 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 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

Algorithm 3: Inversion of Assignment Statements

```
   Routine: InvertAssign(A, s)
1  Let s ≡ y := e
2  foreach x ∈ P do e ← subst(e, x, σ(x))
3  switch U ∩ V(e) do
4    case ∅ s' ← assume(y = e)
5    case {x}
6      s' ← x := LocalInvert(s, x)
7      (A, s'') ← MarkDEmitD(A, {x})
8      s' ← Append(s', s'')
9    otherwise
10     s' ← [ ]
11     x ← Select(U ∩ V(e)); e' ← LocalInvert(s,x)
12     U ← U \ {x}; P ← P ∪ {x}; σ ← σ[x ↦ e']
13   end
14 end
15 return (A, s')
```

Algorithm 4: Transitively Mark Partially Determined Variables as Determined and Emit Assignment Stmts.

```
   Routine: MarkDEmitD(A, W)
1  s' ← [ ]; U ← U \ W; D ← D ∪ W
2  repeat
3    B ← P
4    foreach x ∈ P do
5      if V(σ(x)) ⊆ D then
6        s' ← Append(s', x := σ(x))
7        P ← P \ {x}; D ← D ∪ {x}; σ ← σ[x ↦⊥]
8    end
9  until B = P
10 return (A, s')
```

$Q = \emptyset$ 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 assume 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 $W \subseteq U$ of variables, MarkDEmitD marks them as *determined*. If all variables occurring in the binding expression $\sigma(x)$ of a *partially determined* variable x are now *determined* then x is also *determined* (Line 5). The function MarkDEmitD emits the assignment statement $x := \sigma(x)$. 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 $U \cap V(p)$ 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 $s \equiv$ **if** (p) **then** f_1 **else** f_2 **end**, the function InvertConditional (Algorithm 6) processes p similar to the predicate of an assume statement and inverts the branches f_1 and f_2 individually. Let $A_i = (U_i, D_i, P_i, \sigma_i), i \in \{1, 2\}$. If $P_1 = P_2$ and $\sigma_1 = \sigma_2$ then Join$(A_1, A_2) = (U_1 \cap U_2, D_1 \cup D_2, P_1, \sigma_1)$.

A variable $x \in D \setminus D_1 = U_1 \cap D_2$ is marked as *determined* by the join operation but is *undetermined* in f_1. To force the *determined* status, a random assignment $x := *$ is appended to the inverse of f_1 (Line 8). Similarly, for f_2. 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 $P_1 \neq P_2$ or $\sigma_1 \neq \sigma_2$ then one of the branches is eliminated[1] (Lines 14-17).

In the following program, the variables x_2 and x_3 are *determined* in the if-branch and the else-branch respectively.

```
1 skip;                                  1 ensure(x1, x2: φ(x1, x2));
2 if x1 > 0 then y1 := x2;               2 if x1 > 0 then x2 := y1; x3 := *;
3 else y1 := x3; end                     3 else x3 := y1; x2 := *; end
4 y2 := x3 + x4;                         4 x4 := y2 − x3;
5 y3 := x2;                              5 assume(y3 = x2);
```

(Example 3)

We insert random assignments $x_3 := *$ and $x_2 := *$ in the if-branch and the else-branch respectively to force both x_2 and x_3 to be *determined* along both the branches. We discuss the predicate $\varphi(x_1, x_2)$ later.

Program Exit. Once f is processed completely, the function EmitPRemaining appends random assignments to the *undetermined* variables used in the binding expressions to f'. For each $x \in P$, $x := \sigma(x)$ 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 x_2 gets assigned a value $\neq y_3$ 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 antecedant 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.

[1]It 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 $s \equiv$ assume(p)
2 **foreach** $x \in P$ **do** $p \leftarrow \mathsf{subst}(p, x, \sigma(x))$
3 $(A, s') \leftarrow$ MarkDEmitD($A, U \cap V(p)$)
4 **return** $(A, \mathsf{Append}(s', s))$

$$\frac{\{\psi\} \quad x := e \ \{\varphi\}}{\{\psi \wedge \mathsf{subst}(\varphi, x, e)\} \ x := e \ \{\varphi\}} \, e \neq *$$

$$\frac{\{\psi\} \quad \text{assume}(p) \ \{\varphi\}}{\{\psi \wedge \varphi \wedge p\} \ \text{assume}(p) \ \{\varphi\}}$$

$$\frac{\{\psi\} \quad x := * \ \{\varphi\}}{\{\psi \wedge \varphi\} \ x := * \ \{\varphi\}} \, x \notin V(\varphi)$$

$$\frac{\{\psi\} \quad x := * \ \{\varphi\}}{\{\psi \wedge \varphi\} \quad \text{skip} \quad \{\varphi\}} \, x \in V(\varphi)$$

$$\frac{\{\psi\} \quad s \equiv \textbf{if } (p) \textbf{ then } f_1 \textbf{ then } f_2 \textbf{ end} \quad \{\varphi\}}{\{\psi \wedge ((p \wedge wp(\varphi, f_1)) \vee (\neg p \wedge wp(\varphi, f_2)))\} \quad s \quad \{\varphi\}}$$

$$\frac{f'}{ensure\,(wp\,(true, f'))\,;\, f'}$$

The rule involving $x := *$ and a postcondition φ where $x \in V(\varphi)$ eliminates the random assignment to x as it may violate the postcondition φ. Instead, x is assigned a value that satisfies φ 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: $\varphi(x_1, x_2) \equiv (x_1 > 0 \wedge y_3 = y_1) \vee (x_1 \leq 0 \wedge y_3 = x_2)$. 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_3 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                 1 for i = 1 to 10 do
2   y1[i] := x1[i];                   2   x1[i] := y1[i];
3   y2[i] := x1[i] + x2[i];           3   x2[i] := y2[i] − x1[i];
4 end                                 4 end
```

(Example 4)

Algorithm 7 performs inversion of a loop s. It copies the loop-structure ℓ' of the loop s to the inverse program (Lines 1, 9) but inverts the statements within the loop body

Algorithm 6: Inversion of Conditional Statements

Routine: InvertConditional(A, s)
1 Let $s \equiv$ **if** (p) **then** f_1 **else** f_2 **end**
2 **foreach** $x \in P$ **do** $p \leftarrow \mathsf{subst}(p, x, \sigma(x))$
3 $(A, s') \leftarrow$ MarkDEmitD($A, U \cap V(p)$)
4 $(A_1, f_1') \leftarrow$ InvertSequence(A, f_1)
5 $(A_2, f_2') \leftarrow$ InvertSequence(A, f_2)
6 **if** $P_1 = P_2 \wedge \sigma_1 = \sigma_2$ **then**
7 $A \leftarrow$ Join(A_1, A_2)
8 **foreach** $x \in D \setminus D_1$ **do** $f_1' \leftarrow$ Append($f_1', x := *$)
9 **foreach** $x \in D \setminus D_2$ **do** $f_2' \leftarrow$ Append($f_2', x := *$)
10 $s' \leftarrow$ Append(s', **if** (p) **then** f_1' **else** f_2' **end**)
11 $(A, s'') \leftarrow$ MarkDEmitD($A, \emptyset$)
12 $s' \leftarrow$ Append(s', s'')
13 **else**
14 $A \leftarrow$ Select(A_1, A_2)
15 **if** $A = A_1$ **then**
16 $s' \leftarrow$ Append(s', **if** (p) **then** f_1' **else** assume(false); **end**)
17 **else** $s' \leftarrow$ Append(s',**if** (p) **then** assume(false); **else** f_2' **end**)
18 **end**
19 **return** (A, s')

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 $y[e_0] := x_1[e_1] + x_2[e_2]$. 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 $\{x_1[i'], x_2[i'] \mid 1 \leq i' \leq 10 \wedge i' < i\}$, which describes the set of *determined* locations at the beginning of iteration i. In general, the symbolic representation is of the form $\{S \mid Q\}$ where S is a set of symbolic input array locations with index expressions over some free variables *e.g.*, i, and Q is a constraint over these free variables. Q is of the form $B_i \wedge L$, where B_i is the predicate $(1 \leq i' \leq n_i)$ with n_i as the upper bound of i, and L 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  ℓ' ← GetLoopStructure(s)
2  b ← GetBody(s)
3  U ← U ∪ TopLevelRefs(s) ∪ NestedRefs(s)
4  D ← D ∪ {S | Q}
5  (A, b') ← InvertSequence(A, b)
6  b' ← EmitPRemaining(A, b')
7  φ' ← NonAliasConstr(b') ∧ LoopBoundConstr(b')
8  s' ← assume(φ')
9  s' ← Append(s', InsertBody(ℓ', b'))
10 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 ψ 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.

```
1 for i := 1 to 10 do
2   y[i] := x[i] + x[i + 3];
3 end
```

```
1 x := *;
2 for i := 1 to 10 do
3   x[i + 3] := y[i] − x[i];
4 end
```

(Example 5)

The set of locations *determined* in iterations preceding iteration i is given by $\{S \mid Q\}$, where $Q = (1 \le i' \le 10) \wedge (i' < i)$, 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.

Algorithm 8: Generalized Inversion of Assignments

```
   Routine: InvertAssign(A, s)
1  if s ≡ k := e_k for a local variable k then s' ← s
2  else
3    s' ← [ ]; Let s ≡ y := e
4    foreach r ∈ TopLevelRefs(e) do e ← subst(e, r, σ(r))
5    foreach r ∈ U ∩ (NestedRefs(y) ∪ NestedRefs(e)) do
6      (A, s') ← MarkDEmitD(A, {r})
7    end
8    UV ← ∅
9    foreach r ∈ TopLevelRefs(e) do
10     switch CheckIfUndetermined(r) do
11       case true: UV ← UV ∪ {r}
12       case false: skip
13       case Cond ψ: AddConstraint(ψ); UV ← UV ∪ {r}
14     end
15   end
16   switch UV do
       /* Similar to Algorithm 3, Lines 3-14 */
17   end
18 end
19 return (A, s')
```

The function 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 $\exists i', i \cdot (1 \le i \le n_i) \wedge Q \wedge (\bigvee_{x[e'] \in S} e = e')$ 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.

```
1 for i := 1 to 8 do
2   y[i] := x2[x1[i]];
3 end
```

```
1 ensure(x1 : ∀i · φ(i));
2 for i := 1 to 8 do
3   x2[x1[i]] := y[i];
4 end
```

(Example 6)

The set D of *determined* locations of x_2 at the beginning of iteration i is given by $\{x_2[x_1[i']] \mid (1 \le i' \le 8) \wedge (i' < i)\}$. 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 CheckIfUndetermined symbolically simplifies negation of the above membership test and returns the condition $\psi(i) = \forall i' \cdot ((1 \le i' \le 8) \wedge (i' < i)) \Rightarrow (x_1[i'] \neq x_1[i])$.

The function 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 $\forall i \cdot \varphi(i) = \forall i \cdot (1 \leq i \leq 8) \Rightarrow \psi(i)$. We call these constraints *non-aliasing constraints*. The function NonAliasConstr 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 encouter 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 $sum(p, q)$ to indicate array sizes. $sum(p, q)$ is summation of p, q times. The arguments to sum can be input variables including array subscripts. Consider the following program.

```
   Input: int n, m, x[sum(n, m)];
   Output: int n', m', y[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 $sum(n, m) = sum(n', m')$. 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.

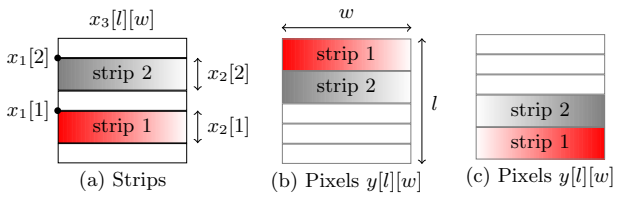

Figure 3: Essentials of Strip-based TIFF Images

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.

```
1 typedef struct _tiff {
2    int orientation, n, w, r;
3    int x1[N], x2[N], x3[N][N];
4 } tiff;
```

(a) TIFF data structure

```
1 int y[N][N];
```

(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

```
   Input: int n, w, r, x1[n], x2[n], x3[sum(n, x2[i])][w];
   Output: int n', w', x2'[n'], y[sum(n, x2[i])][w];
 1 switch orientation do
 2   case 1 /* left-to-right, top-to-bottom */
 3     m := 1;
 4     for i := 1 to n do
 5       for j := 1 to x2[i] do
 6         for k := 1 to w do
 7           assume((i < n ⇒ x2[i] = r) ∧ (x2[n] ≤ r));
 8           y[m][k] := x3[x1[i] + j][k];
 9         end
10         m := m + 1;
11       end
12       x2'[i] := x2[i]; /* instrumentation */
13     end
14     n' := n; w' := w; /* instrumentation */
15   case 2 /* right-to-left, top-to-bottom */
16     ... y[m][k] := x3[x1[i] + j][w − k] ...
17   case ...
18 end
```

Figure 4: Normalization Program for TIFF Images

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. Figures 3(b)-(c) indicate canonical representation of a **tiff** data for left-to-right top-to-bottom and right-to-left bottom-to-top orientations respectively. We indicate the width and the length of an image by w and ℓ respectively. ℓ is not required to be part of the **tiff** data structure. The normalization program for **tiff** is given in Figure 4. For brevity, we use the switch-case construct and do not give code for all 8 possible orientations which differ only in the innermost assignment statement. We also directly refer to elements of the **tiff** data structure instead of the usual struct notation.

The number of rows of all strips (except possibly the last) should be equal to a rows per strip field r as indicated by the assume statement within the loop. It is possible to simplify the definition of **tiff** to remove array x_2 and instead use a scalar variable to indicate $x_2[n]$. This however does not make much difference to the overall effectiveness of inversion and testing. Note that the dependent type signature and instrumentation are used to specify loop-bound constraints.

4.2 Implementation and Results

Our prototype implementation inverts the normalization program and generates the following constraint for Case 1 (Figure 4) . The constraints for other cases are similar and solved by ensure() when the inverse program is executed.

$$sum(n, x_2[i]) = sum(n', x_2'[i]) \wedge w = w' \quad (1)$$

$$\wedge\ (\forall i \cdot 1 \le i < n \Rightarrow x_2[i] = r) \wedge (x_2[n] \le r) \quad (2)$$

$$\wedge\ (\forall i, j, k, i', j', k' \cdot (i', j', k') \prec (i, j, k) \quad (3)$$

$$\Rightarrow (x_1[i] + j \neq x_1[i'] + j') \vee (k \neq k')) \quad (4)$$

The constraints (1) are the loop-bound constraints. 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:

$$\forall i \cdot (x_1[i] + x_2[i] \le x_1[i+1]) \vee \forall i \cdot (x_1[i+1] + x_2[i+1] \le x_1[i])$$

where $1 \le i < n$. After instantiating n to, say κ, the loop-bound constraint $sum(n, x_2[i]) = sum(n', x_2'[i])$ is expanded to $x_2[1] + \ldots + x_2[\kappa] = x_2'[1] + \ldots + x_2'[n']$. 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 $\kappa - 1$. 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.

Application	Orientation	Rows per strip	Layout
FastStone 3.6	✓	✓	✓
KView 3.5.4	✓	✓	✓
GIMP 2.2.13	✗	✓	✓
Open Office 2.0.4	✗	✓	✓
Picasa 3.6	✗	✓	✗

Figure 5: Test Results for TIFF Clients

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 ✓ indicates that the application passed all test cases and ✗ 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 (✓ above) are more plausible.

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 abstract 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 representation 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 functionality of grammar-driven software by enumerating test data from a grammar definition. In our work, the exploration of the syntactic space of data is driven by data equivalence specification in terms of normalization programs. Korat [8] systematically enumerates data instances from a data structure invariant specified as a Java predicate. Test case generation for representation dependence is a technically different 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 inputs and outputs of the program. In contrast, our test generation 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 practice. 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 problem 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

[1] http://mail.python.org/pipermail/image-sig/1999-may/000730.html.

[2] http://www.asmail.be/msg0055369928.html.

[3] http://www.zan1011.com/tiff.htm.

[4] www.gamedev.net/reference/articles/article538.asp.

[5] Adobe Dev. Assoc. TIFF Revision 6.0, June 1992.

[6] A. Banerjee and D. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM*, 52:894–960, 2005.

[7] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, 2004.

[8] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *ISSTA*, pages 123–133, 2002.

[9] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, 2003.

[10] J. Boyland. Why we should not add readonly to Java (yet). *J. of Object Tech.*, 5(5):5–29, June 2006.

[11] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Hong Kong University of Science and Technology, 1998.

[12] D. G. Clarke, J. Noble, and J. Potter. Overcoming representation exposure. In *Proc. of the Work. on Object-Oriented Tech.*, pages 149–151, 1999.

[13] D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.

[14] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[15] E. W. Dijkstra. Program inversion. In *Program Construction*, volume 69 of *LNCS*, pages 54–57, 1978.

[16] B. Dutertre and L. De Moura. The Yices SMT solver. Technical report, SRI, 2006.

[17] Y. Ge, C. W. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *Ann. Math. Artif. Intell.*, 55(1-2):101–122, 2009.

[18] A. Gotlieb and B. Botella. Automated metamorphic testing. In *COMPSAC*, page 34, 2003.

[19] D. Gries and J. L. van de Snepscheut. Inorder traversal of a binary tree and its inversion. *Formal Development of Programs and Proofs*, pages 37–42, 1990.

[20] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *OOPS Mess.*, 3(2):11–16, 1992.

[21] http://www.libtiff.org.

[22] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Soft.*, 7(4):50–55, 1990.

[23] C. Murphy, K. Shen, and G. Kaiser. Automatic system testing of programs without test oracles. In *ISSTA*, pages 189–200, 2009.

[24] B. J. Ross. Running programs backwards: The logical inversion of imperative computation. *Formal Asp. Comput.*, 9(3):331–348, 1997.

[25] B. Schoenmakers. Inorder traversal of a binary heap and its inversion in optimal time and space. In *MPC*, volume 669 of *LNCS*, pages 291–301, 1992.

[26] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute force vulnerability testing.* Addison-Wesley, 2007.

[27] M. Trauth. *MATLAB Recipes For Earth Sciences.* Springer, 2006.