Automating Traceability for Generated Software Artifacts

Julian Richardson
RIACS/USRA
Automated Software Engineering Group
NASA Ames Research Center, Moffett Field, CA 94035-1000, USA
julianr@email.arc.nasa.gov

Jeff Green
QSS Inc
jeffreyadamgreen@yahoo.com

Abstract

Program synthesis automatically derives programs from specifications of their behavior. One advantage of program synthesis, as opposed to manual coding, is that there is a direct link between the specification and the derived program. This link is, however, not very fine-grained: it can be best characterized as `Program is-derived-from Specification`. *When the generated program needs to be understood or modified, more fine-grained linking is useful.*

In this paper, we present a novel technique for automatically deriving traceability relations between parts of a specification and parts of the synthesized program. The technique is very lightweight and works — with varying degrees of success — for any process in which one artifact is automatically derived from another.

We illustrate the generality of the technique by applying it to two kinds of automatic generation: synthesis of Kalman Filter programs from specifications using the AutoFilter program synthesis system, and generation of assembly language programs from C source code using the GCC C compiler. We evaluate the effectiveness of the technique in the latter application.

1. Introduction

One way in which traceability information can be derived is to augment the program synthesis system so that manipulations and calculations it carries out during the synthesis process are annotated with information on what the manipulations and calculations were and why they were made. This information is then accumulated throughout the synthesis process, at the end of which, every artifact produced by the synthesis is annotated with a complete history relating it to every other artifact (including the source specification) which influenced its construction. This approach requires modification of the entire synthesis system — which is labor-intensive and hard to do without influencing its behavior.

We describe a lightweight, technique for deriving traceability from a program specification to the corresponding synthesized code. Once a program has been successfully synthesized from a specification, small changes are systematically made to the specification and the effects on the synthesized program observed. The technique builds on work first described in [RG]. In this paper we describe how the technique has been completely automated, and present an evaluation of the results.

We have applied the technique to one of our program synthesis systems, AUTOFILTER, and to the GNU C compiler, GCC. The technique was partially automated for the AUTOFILTER application and fully automated for the GCC application. The results are promising:

1. Inspection of the results indicates that in semiautomatic experiments with our synthesis system, most of the connections derived from the specification to the synthesized code are correct, and around half of the lines in the synthesized code can be traced back to at least one line of the specification.
2. In the GCC experiments, 75% of the traceability links derived using automatic perturbation involving copying were correct. 20-40% of the lines in the generated assembler could be correctly traced back to the C source program.
3. Small changes in the source often (especially in the GCC examples) induce only small changes in the tar-

get.

2. Program Generation and Traceability

Traceability from requirements through to program code provides a rationale for the code. There are many reasons why this is desirable, of which some are:

- It provides an aid to understanding the code.
- It provides an aid to understanding the requirements.
- It provides an aid to understanding why the code does or does not work correctly. This is particularly important in safety and mission critical applications.

In practice, traceability can be hard to achieve when human programmers are involved. Programmers are reluctant to maintain documentation, and traceability is easily broken if programming artifacts (requirements, design elements, documents, code etc) are altered without making corresponding changes to the other programming artifacts which they should affect or be affected by.

Program synthesis is a technique for automatically deriving programs from specifications of their behavior. A good specification language enables requirements to be stated in a natural way. Program changes can be realized entirely as changes to the program's specification.

The Automated Software Engineering Group at the NASA Ames Research Center has been researching and building domain-specific program synthesis systems (recently, AUTOBAYES [FS02], AUTOFILTER [WVS+01] and before that AMPHION [LPPU94]). Since program synthesis systems are in general large and complex, and therefore not necessarily entirely trustworthy, part of our research has addressed the synthesis of non-code artifacts which provide evidence that the synthesized programs correctly implement their specifications. In particular, the group has been developing:

- extensible *automatic certification* of synthesized programs [WSF02, DF03] — the synthesis system synthesizes code annotations along with the program code, and these annotations are used to guide a theorem prover to prove certain safety properties.
- *automatic documentation* of synthesized programs [WVS+01] — program documentation is synthesized at the same time as the program code.

Traceability information is another kind of non-code information which provides evidence of a program's fitness for its task.

In the following sections we outline two techniques by which this traceability information can be automatically derived. The first technique, which we will call in this paper *deep traceability*, involves augmenting the program synthesis system (including program schemas and axioms) so that calculations carried out by the synthesis system are annotated with information on what the calculations were and why they were made. We concentrate in this paper on describing a second technique, which we call *surface traceability*, which is novel and lightweight; once a program has been successfully synthesized from a specification, small changes are systematically made to the specification and the effects on the synthesized program observed.

A note regarding notation: we call the input to the program generation process the *source*, and the output the *target*. In the case of a program synthesis system, the source is a specification, and the target is a program (C code, for example). In the case of a compiler, the source is a (C) program, and the target is an assembly language program.

3. Deep Traceability

A technically sound but heavyweight approach to tracing from specifications to generated programs involves augmenting the program synthesis system (including program schemas and axioms) so that calculations carried out by the synthesis system are annotated with information on what the calculations were and why they were made. This approach was adopted in the *ExplainIt!* extension of AMPHION/NAV [WVS+01]. AMPHION/NAV is a purely deductive synthesis system, which extracts programs from proofs carried out in a tableau style theorem prover. The proofs can be structured into trees whose nodes are sets of formulae, and an edge exists links two nodes if the first is derived from the second. Explanations are attached to the axioms in AMPHION/NAV's domain theory, propagated along the edges in the derivation tree, and finally incorporated into an XML document which links each program statement to the axioms and parts of the program specification involved in its construction.

The approach works well for a purely deductive synthesis approach but requires extensive modification of the entire synthesis system. For complex third-party code generators, for example a C compiler, the deep traceability approach is not practicable.

In the rest of this document, we describe a new technique which can trace complex relationships between source and target and requires very little effort to implement.

4. Surface Traceability

We discover, automatically, relationships between source and target in the following way: first, the synthesis system (or compiler) is applied to the source to generate the target. We call the original source the *nominal source* and the corresponding generated target the *nominal target*. Next, small changes (we call them *perturbations*) are made (one at a time) to the source (yielding a *perturbed source*), and corresponding target programs are synthesized (or compiled) from it (resulting in either failure, or in a *perturbed target*). As long as the synthesis process is deterministic, differences between the nominal and perturbed target programs can only be caused by the differences between the nominal and perturbed sources. We therefore associate lines in the nominal target program which differ in a perturbed target program with the lines in the nominal source which were changed by the perturbation. An example will demonstrate how the technique works, as well as its flexibility.

Consider a system which automatically synthesizes English sentences from corresponding French specifications. For our current purposes, assume that one word of source (or target) is written per line of input (or output). Let the nominal source be *"Ceci n'est pas une pomme."* From this we synthesize (using an automatic language translator, for example) the nominal target, *"This is not an apple."* Apply separately the perturbations *pomme* → *banane*, *pas* → *pipe*, and *une* → *la*, resulting in *"This is not a banana."* for the first perturbation, an error for the second, and *"This is not the apple."* for the third perturbation. We associate the differences between the perturbed and nominal targets with the corresponding perturbations, in this case we associate "apple" with "pomme" and "an" with "une". The technique did not require any knowledge of English or French, apart from the ability to make localized changes in the (French) source and detect differences induced in the (English) target.

The main advantages of the proposed technique are all closely related:

1. It is very lightweight: it is extremely simple to implement, and quite effective. In our initial implementation (§5), perturbations are applied by a line editor, and differences are determined by the Unix `diff` program.
2. It does not require modification of the synthesis system (or compiler). This greatly reduces the effort needed to employ it, removes the possibility of inadvertently introducing errors into the synthesis system when it is modified, and makes the technique applicable to third-party code generators such as compilers.
3. It does not require detailed, or indeed any knowledge of the internal mechanisms of the synthesis system, which is treated as a black box.

There are of course disadvantages, which we note here:

1. It cannot identify every part of the source which influences the target.
2. Some small changes in the specification can appear to have profound effects when in fact the synthesized programs are equivalent. For example, a variable name which occurs in many lines of the program might be changed. Note that this effect would also appear in a deep traceability approach unless measures were taken to overcome it, for example by developing a notion of α-equivalence (in the sense of the λ calculus) for the generated programs.
3. Some changes cannot be made without also making other corresponding changes. For example, to discover the effect on the target of the name of a function which is declared in the source, *all* lines in the source which contain that function name have to be changed simultaneously, or an error will result. We therefore cannot discover the effect of function naming with only single-line changes to the nominal specification.
4. It requires careful choice of which perturbations to apply to the source specification/program. We describe how this choice can be automated in §7.

5. Initial Implementation

Manual experiments indicated that the technique might be interesting, so we decided to automate it. The system is used as follows:

- A list of perturbations is given to the system, expressed as commands (the *perturbation* `ed` *commands*) for the Unix *ed* editor. Each perturbation only alters a single line in the source.
- For each perturbation, a shell script applies the following steps:
  - The perturbation is applied to the nominal specification to obtain a *perturbed specification.*
  - The synthesis system is applied to the perturbed specification, either failing, or yielding a perturbed program.
  - If synthesis failed, this is noted in a log file, otherwise the differences between the perturbed program and the nominal program are computed (using Unix `diff -w`) and appended to the log file.
- Some irrelevant information is removed from the log file (leaving for each change the specification line changed followed by the `ed` commands produced by `diff` which describe the difference (if any) between the corresponding perturbed and nominal programs).
- A number of `emacs` macros are used to:
  - Remove differences which only add lines to the nominal program — we exclude these since we are going to annotate the nominal program with the changes and in this case the lines which are added do not exist in the nominal program, only in the perturbed program.

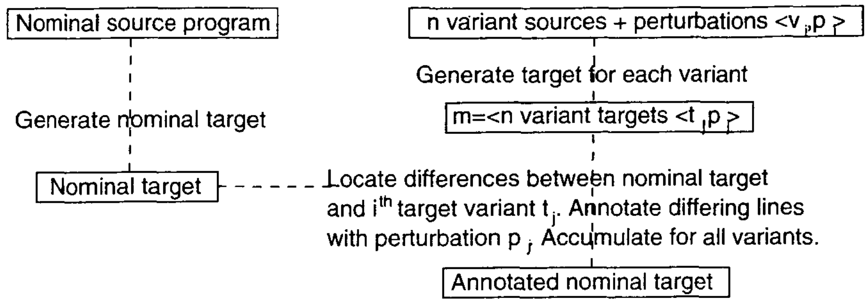
Nominal source program
Generate nominal target
Nominal target
n variant sources + perturbations <v ,p >
Generate target for each variant
m=<n variant targets <t ,p >
Locate differences between nominal target
and i^th target variant t_i. Annotate differing lines
with perturbation p_i. Accumulate for all variants.
Annotated nominal target

Figure 1: Generation of annotations. Each v_i is a variant program and the corresponding p_i is a set of the line numbers of lines of the differences between v_i and the nominal source program.

  - Move perturbations which produced no effect (or only changed a date stamp in the generated target) into a separate file.
  - For each remaining difference, derive an ed command which will append the perturbation ed commands to the lines in which program which they affect.
- These derived ed commands are finally applied to the nominal program, yielding the *annotated program*, in which each line may be annotated with one or more perturbation ed commands, corresponding to the perturbations which affected that line in the program (as judged by that line differing in the perturbed program from the nominal program).

6. Initial Results

In this section we describe the results of applying our technique in two contexts: the AUTOFILTER program synthesis system, and the GNU GCC compiler.

6.1. AUTOFILTER

Initially, the technique was manually applied to an AUTOFILTER specification (a simplified specification of part of the Deep Space 1 probe's attitude control system). The specification has 134 lines (of which 44 are non-blank, non-comment lines). The nominal program has 362 lines (of which 235 are non-blank, non-comment lines). A total of 37 perturbations were manually applied. 9 led to failed synthesis attempts, 9 did not lead to any changes in the synthesized program, 6 changed only temporary variable names in the generated code (the programs were α-equivalent), and 10 reveal interesting relationships between the source and target.

In the first semiautomated experiment, using the same specification, 67 perturbations were chosen: 18 led to failed synthesis attempts, 19 did not lead to any change in the synthesized program. The remaining 30 generated annotations of the synthesized program. Of these 30, 6 changed many lines in the target, changing the number or order of input variables to the synthesized code, or the size of its internal matrices and vectors. In total, 143 non-blank, non-comment lines in the generated code were annotated.

In the second semiautomated experiment, applied to an AUTOFILTER specification for thruster control during automated docking (source: 143 non-empty, non-comment lines; output: 220 non-blank, non-comment lines), 43 perturbations were applied. 16 led to failed synthesis attempts, 6 did not lead to any changes in the synthesized program, 9 led to localized changes, 9 led only to temporary variable name changes, and 3 changed many lines in the target.

Manual inspection of the annotated target programs produced in the two semiautomated experiments suggest that most of the lines in the synthesized program can be traced back to one or more lines in the specification, that the relationships identified between specification and synthesis program are correct, as judged by someone who understand the meanings of the specifications and the synthesized programs.

We now present a more detailed example.

6.2. GCC

In order to demonstrate the flexibility of our technique for surface traceability, we applied it to the generation of assembly language code from C source code. Figure 2 shows the source program, and figure 3 shows the annotated assembly language program which was generated. In order to fit space requirements the information has been manually edited: only the main section of the generated assembly language code is shown, each perturbation has been written as the source code line number to which it applied and a letter, listed at the beginning of the assembler line which it traced. The perturbations have been shown directly in the source program listing. Only perturbations which traced lines in the main section of assembler code in figure 3 are shown. Others either had no effect, caused an error, or affected a part of the assembler code outside the main section.

The resulting annotated assembly code identifies many relationships between it and the C source code. Here is our interpretation of some of the results: first, note that most perturbations only affect a small number of lines in the generated assembler. The exceptions to this are perturbations 8A and 10E which change many lines of the generated assembler code (probably because in changing the datatypes which represent i and a they affect register allocation and memory offsets although we can't conclude this from our experiments — to draw this conclusion probably requires some knowledge of assembler and the amount of memory needed to store ints versus doubles versus floats). Perturbation 27H also results in a significant change, possibly for a similar reason. Perturbations 16B, 16C, 16R identify those parts of the target associated with the head of the `for` loop. Perturbations 34L and 37M trace the call to the *exp* function. Perturbation 30P traces the assignment of the result to y. Perturbation 40Q traces where y is printed. Perturbation 16R traces that add instruction to the loop header. Other relationships between source and target are made evident by our experiment: readers are invited to determine these themselves.

7. Automatic Generation of Perturbations

7.1. Introduction

In previous sections we showed how our technique of applying perturbations to a source and detecting how those perturbations affect the target can be used to annotate the target with traceability information connecting some lines of the target program with corresponding lines of the source. The technique was automatic apart from a few steps: calling a few appropriate scripts and macros, and specifying the perturbations. Automating the first of these is not difficult — it requires writing one more script which calls the right scripts at the right times. The second is more fundamental in character. In this section we show how perturbations can be automatically generated. Derivation of traceability information is then essentially automatic. We apply the technique to a simple C program, employing several different kinds of perturbation. We evaluate the resulting traceability information in terms of accuracy: what percentage of traceability links correctly relate target statements to corresponding source statements, and coverage: what percentage of target statements are correctly related to any source statement.

Figure 4 illustrates how the perturbations are generated. We outline the steps here and describe them in more detail in the following sections.

Generating slight variants of a given source program is not trivial. The simplest automatic approach: randomly adding, deleting etc characters and sequences of characters from the program is impractical — the vast majority of the resulting programs would be syntactically ill-formed. Ill-formed programs provide us with no traceability information. If we have a more structured representation of the program, however, we can limit the kinds of changes we make to those which are likely to produce well-formed programs. For this reason, the first step of the automatic perturbation process is to parse the program into a structured program representation — in our case the AUTOBAYES Intermediate Language is a convenient choice. The result is a Prolog term (T, say). Well-defined C expressions and statements of different kinds are represented by subterms with different functors. For example, an assignment `v=e` is represented by a Prolog term $assign(v', e')$ (where v', e' are the AUTOBAYES Intermediate Language representations of the C expressions `v` and `e` respectively). The parsed program T is run through the AUTOBAYES code generator to generate the nominal source program. The nominal source program may differ in terms of formatting from the original source program but should otherwise be functionally identical.

The AUTOBAYES Intermediate Language term T is then perturbed in several different ways: by *swapping* two subterms of T, by *copying* one subterm of T to another location in T, by *deleting* a subterm of T, or by a *small copy*, which is a copy restricted to only copying single operators to single operators, or single constants to single constants, or single variables to single variables. Some effort is made to ensure that these operations produce syntactically correct AUTOBAYES Intermediate Language terms. Each of these perturbed versions of T is then run through the AUTOBAYES code generator. Each successful code generation yields a variant program which differs slightly from the nominal source program. The differences (line numbers) are localized using `diff -w` and associated with the variant.

The end product of this process is a set $\langle v_i, p_i \rangle$ of variant programs v_i and associated differences p_i (which are sets of line numbers). Traceability information can then be generated from these using the technique described in §5.

7.2. Parsing the Source

A small parser for C was written using a Prolog DCG (approximately 200 lines of code including comments). Programs are parsed into terms in the AUTOBAYES Intermediate Language. The grammar suffices for the purposes of experimentation, but no attempt has been made to cover the whole C language. Some limitations are imposed by the AUTOBAYES Intermediate Language. For example, `for` loops have a restricted form and a limited range of data types is supported. Types `int`, `bool`, `double`, `void` are supported, as scalars, 1- and 2-dimensional ar-

```
                                      #include <stdio.h>
                                      #include <math.h>

                                      int main()
                                      {
                                      double t, tf,
                                      x, y;
8A int → double                       int i;

10E double → float                    double a = 60,
                                      b = 0.0782,
                                      kappa = 1.95,
13G 0.5 → 1                           c = 0.5;
14H t → t+1                           tf = 1.0/5.0;

16B 0 → 1 16C < → > 16R ++ → --       for(i=0; i < 100; i++)
                                      {
18D tf → t                            t = i * tf;

                                      /*
                                      y = a*exp(-b)...
                                      x = c*kappa*a*...
                                      */

25F c → kappa                         x = c*
                                      kappa*
27H t → t+1                           ((1-pow(kappa,t))/
                                      (1-
29I kappa → c                         kappa));
30P y → x                             y = a*
                                      exp(
32J b → b-1                           -b)*
33N 1 → 2                             ((1-
34L exp → log                         exp(
35K b → kappa                         -b*
                                      t))/
37M exp → log                         (1-exp(
                                      -b)));

40Q x → kappa                         printf("%f %f ", x, y);
                                      }}}
```

Figure 2: The C source code, and a list of the perturbations which applied to the section of assembly language code in figure 3. Note that since our technique traces target lines of code to source lines of code, we have split compound statements into multiple lines.

rays. Variables must be declared before the body of the program code. Only the following high level constructs are supported: conditionals `if ... then`, `if ... then ... else`, and loops of the form `for (<init>, <e1> <= <e2>, <v>++) <stmt>` where `<init>` is an assignment, `<e1>` and `<e2>` are expressions (i.e. the loop test must be "less than or equal to"), <v> is any C variable and `<stmt>` is a statement (either a single statement or a block enclosed in {...}). Other limitations exist, for example a part of the grammar is left recursive causing parsing of some expressions not to terminate.

The result of parsing is an AUTOBAYES Intermediate Language Prolog term representing the source program.

7.3. Generation of Perturbations

Auxiliary predicates *getsub(+Term,?Pos,?SubTerm)* and *setsub(+Term,?Pos,?NewSubTerm,?NewTerm)* are defined for accessing and modifying AUTOBAYES Intermediate Language terms and subterms. A variant of an AUTOBAYES Intermediate Language term T is generated as follows:

```
8A 16B              st %g0, [%fp-52]
8A                  .LL3: ld [%fp-52], %o0
8A 16C              cmp %o0, 99; ble .LL6
                    nop; b .LL4; nop
8A                  .LL6: ld [%fp-52], %f4; fitod %f4, %f2
18D                 ldd [%fp-32], %f4; fmuld %f2, %f4, %f2
                    std %f2, [%fp-24]
10E                 ldd [%fp-80], %o0
27H                 ldd [%fp-24], %o2
                    call pow, 0; nop; fmovs %f0, %f4; fmovs %f1, %f5
10E 25F             ldd [%fp-88], %f2
10E                 ldd [%fp-80], %f6; fmuld %f2, %f6, %f2
8A 13G              sethi %hi(.LLC5), %o1; or %o1, %lo(.LLC5), %o0
                    ldd [%o0], %f6; fsubd %f6, %f4, %f4
8A 10E 13G          sethi %hi(.LLC5), %o1; or %o1, %lo(.LLC5), %o0
10E                 ldd [%o0], %f6
10E 29I             ldd [%fp-80], %f8
10E                 fsubd %f6, %f8, %f6; fdivd %f4, %f6, %f4
                    fmuld %f2, %f4, %f2; std %f2, [%fp-40]
10E 32J             ldd [%fp-72], %f2
10E 27H 32J         fnegs %f2, %f4; fmovs %f3, %f5; std %f4, [%fp-16]
                    ldd [%fp-16], %o2; mov %o2, %o0; mov %o3, %o1;
                    call exp, 0; nop
10E                 std %f0, [%fp-96]
10E 35K             ldd [%fp-72], %f4
10E                 fnegs %f4, %f2; fmovs %f5, %f3; ldd [%fp-24], %f4
10E 27H 32J         fmuld %f2, %f4, %f6; std %f6, [%fp-16]
                    ldd [%fp-16], %o2; mov %o2, %o0; mov %o3, %o1
34L                 call exp, 0
                    nop
10E                 std %f0, [%fp-104]; ldd [%fp-72], %f2
10E 27H 32J         fnegs %f2, %f8; fmovs %f3, %f9; std %f8, [%fp-16]
                    ldd [%fp-16], %o2; mov %o2, %o0; mov %o3, %o1
37M                 call exp, 0
                    nop; fmovs %f0, %f2; fmovs %f1, %f3
10E                 ldd [%fp-64], %f6
10E 27H 32J         ldd [%fp-96], %f10; fmuld %f10, %f6, %f4
8A 13G 33N          sethi %hi(.LLC5), %o1; or %o1, %lo(.LLC5), %o0
                    ldd [%o0], %f8
10E                 ldd [%fp-104], %f10
                    fsubd %f8, %f10, %f6
8A 13G              sethi %hi(.LLC5), %o1; or %o1, %lo(.LLC5), %o0
                    ldd [%o0], %f8; fsubd %f8, %f2, %f2; fdivd %f6, %f2, %f6;
                    fmuld %f4, %f6, %f2
30P                 std %f2, [%fp-48]
8A 10E 13G 33N      sethi %hi(.LLC6), %o1; or %o1, %lo(.LLC6), %o0
40Q                 ld [%fp-40], %o1; ld [%fp-36], %o2
                    ld [%fp-48], %o3; ld [%fp-44], %o4; call printf, 0; nop
8A                  .LL5: ld [%fp-52], %o0
16R 8A              add %o0, 1, %o1
8A                  st %o1, [b .LL3
```

Figure 3: The annotated assembly language code.

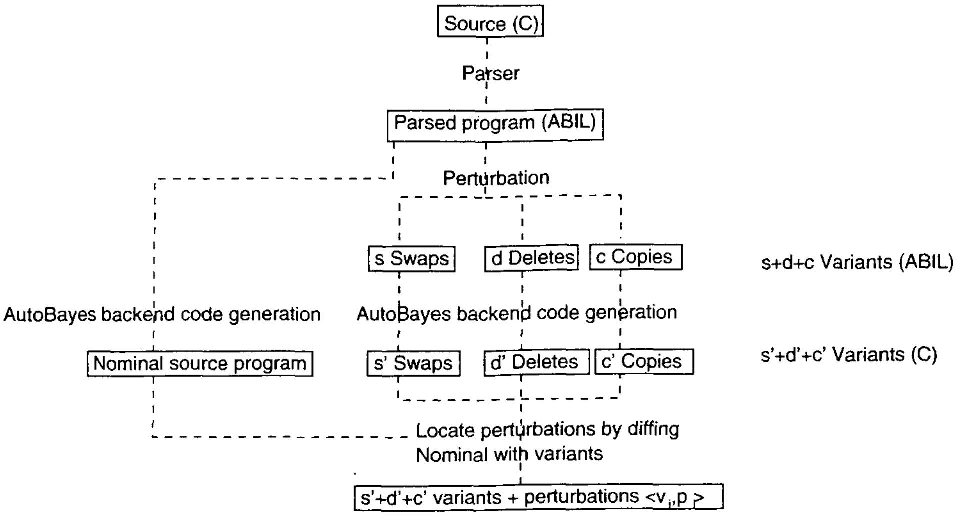

Figure 4: Automatic generation of perturbations. Boxes represent artifacts, lines represent processes. s, d, c, s', d', c' represent numbers. The input to the perturbation process is a single source C program p. The output is a set of pairs $\langle v, d \rangle$ where each v is a variant of p and d a list of the lines which differ between p and v.

- Generate a list Ps containing the position of every subterm of T. Let L be the length of the list Ps.
- Generate two random integers I, J, uniformly distributed in the range $[0, L)$. From these generate two random positions P, Q, respectively the I^{th} and J^{th} (counting from zero) elements of Ps. Let T_P and T_Q be the subterms of T at positions P and Q respecively.
- If the user has not specified a particular operation to be applied, randomly pick one of *swap*, *copy*, *delete* or *small copy*.
- Perform the following action, depending on the chosen operation:
  - *swap* — return the term formed by setting position P of T to T_Q and position Q of T to T_P,
  - *copy* — return the term formed by setting position P of T to T_Q,
  - *small copy* — a copy restricted to only copying single operators to single operators, or single constants to single constants, or single variables to single variables (e.g. replacing a "+" with a "-", or "1.002" with "-23" or "v" with "kappa").
  - *delete* — return the term formed by setting position P of T to $skip$, 0 or $true$ depending on whether T_P is a statement, expression or boolean.

The result of the process is to produce a number of variants of the AUTOBAYES Intermediate Language term T.

7.4. Code Generation

AUTOBAYES's code generator back end produces C language files from terms in the AUTOBAYES Intermediate Language. It can also generate C++ code suitable for compilation with a number of different libraries, for example *Octave* and *MATLAB*. In this application we generate standalone C code.

In order to ensure that the nominal source program and the perturbed programs do not differ in formatting or use of comments, the AUTOBAYES Intermediate Language term T obtained by parsing the original source program is fed through the code generator, resulting in a C source file which may differ from the original source file in formatting, but is otherwise functionally identical.

Each variant AUTOBAYES Intermediate Language term generated by the perturbation process above is fed in turn through the code generator. Usually, some of the variants (around 50%) do not conform to legal AUTOBAYES Intermediate Language syntax and do not correspond to legal C programs. The code generator fails to generate code from these. The variants which are syntactically valid each produce a single C program.

7.5. Localization of Differences

In §5, lines of the nominal target were annotated with the *perturbation commands*, which served both to specify both the line number of the perturbation and what substitution was applied to derive the perturbation. In the automatically derived perturbations there is not necessarily a succinct way (such as a substitution) to specify exactly how the perturbed source was derived from the nominal source. We therefore omit that information and only annotate the nominal target with the line numbers of lines affected by the corresponding perturbation. These line numbers are derived by

- comparing the perturbed C program with the nominal C program using `diff -w`.

7.6. Results

We applied the automatic perturbation and subsequent traceability analysis to a simple C program which contained a number of features: two `for` loops, floating point arithmetic, an `if ... then ... else` statement. Four experiments were carried out using only *swaps*, only *deletes*, only *copies*, and only *small copies*. Two additional experiments were performed: one employing the *copy* technique with different (automatically generated) perturbations, and one employing the *delete* technique with one of the perturbations from the original *delete* experiment excluded. The traceability links for the resulting six annotated assembly language programs were evaluated according to:

- Accuracy: what percentage of the annotated lines correctly linked the nominal target to the corresponding C source program statement?
- Coverage: what percentage of the lines in the nominal target were correctly linked to a line in the source C program?

Where a line in the nominal target is linked to more than one line in the nominal source, the linkage is deemed incorrect if any of the links are incorrect.

The results of the experiments are summarized in the table below. The first row gives the kind of operation performed, the second and third rows give the accuracy and coverage. Figures for accuracy are number of correct links/total number of links. Figures for coverage are number of correct links/total number of lines in nominal target. The additional experiments are labeled *copy2* and *del2*. Each experiment accumulated the result of 15-20 perturbations.

opn	copy	copy2	swap	del	del2	small
acc	78/88	45/59	1/102	1/137	53/65	39/45
cvg	78/179	45/179	1/179	1/179	53/179	39/179

Results for the *copy* operations were good: more than 75% of the derived traceability links were correct. Results for the *swap* and *delete* operations were bad. A significant reason that the result of the *swap* operations is so astoundingly poor is that each *swap* produces *two* differences between the nominal and perturbed source, and each line in the nominal target will in general be linked to one of those — so one will be wrong, and the link will be deemed incorrect. The failure to derive useful information from the first experiment using *delete* operations is caused by one of the automatically generated perturbations, which deletes the entire body of the outer `for` loop, thereby affecting a large number of lines in the resulting compiled program and causing much of the nominal target program to be annotated with every line of that body. When that perturbation is excluded (in experiment *del2*), the results are comparable to those obtained using copying.

The results indicate that the automatically generated perturbations using the *copy* and *smallcopy* operations are effective. The *swap* operation is not useful for generating traceability information. The *delete* operation may be useful as long as perturbations which delete large sections of code are avoided.

Coverage may be improved by applying more perturbations in each experiment.

8. Related Work

The most closely related work is in program understanding, although little work in that area has not been applied to automatically generated artifacts. In [CE01], a technique is presented which analyzes assembler code to locate jump tables (which arise when compiling `case` statements). The technique is based on slicing and analysis of use of memory locations and registers in the code. It may be possible to use our shallow traceability technique to locate jump tables. In [ACPT01], traceability links from a design document or artifact to its a program which implements it are generated by locating similarities between the design and implementation.

9. Applications

We envisage the following applications of the derived traceability information:

- Facilitate understanding of a synthesized program (or assembler produced by a compiler, or some other automatically generated artifact). This is important in cases where the user of a synthesis system may not understand or trust the synthesis process, or when the synthesized program needs to be manually reviewed or edited.
- Ensure that requirements are (correctly) implemented. Some lines in the specification are not linked to any lines in the synthesized program because changing them does not affect the synthesized program. If these specification lines correspond to parts of important requirements, then we may have identified a problem.
- Determine the effects of parts of the specification. For example, in AUTOFILTER we can specify approximations which may be applied to the synthesized program

(e.g. for efficiency reasons). The traceability information we derive can pinpoint the effects, if any, of these approximations.

10. Further Work

Our experiments have shown that the proposed technique can be automated and can correctly link source lines to related target lines. The technique can be accurate but in each of our experiments coverage was relatively low, i.e. many lines in the generated target could not be traced back to any line in the source. It is possible to improve coverage by accumulating the results of more perturbations in each annotated target, but we expect this will accumulate erroneous as well as correct links. The figures in our evaluation table might then get worse. One interesting possibility would be apply a very large number of perturbations and to generate *probabilities* for each link — a link which is generated by 400 different perturbations is more credible than one which is generated by 3. Erroneous links may then be less of a problem.

In the AUTOFILTER example, many of the lines in the synthesized program were marked as changed merely because a variable name had changed. This suggests that a more sophisticated way of determining differences, which takes account of unimportant changes in variable names (i.e. α-equivalence) would be useful. Similarly, in the GCC example, some changes affected a large number of lines in the generated assembler code because they changed memory addresses or register allocation.

The annotated target produced by the system is adequate for experiments but a better form of output would be useful, for example lines in the annotated target could be linked using HTML to the lines in the source which affected them or presented in the form of a traceability matrix.

The technique needs to be evaluated on more complex programs and specifications.

11. Conclusions

The technique we have presented, though extremely simple, has the power to discover relationships between source and target which would otherwise require detailed knowledge of source and target languages, the meaning of the source and target programs, or the program generation (compilation) mechanism itself. We successfully leverage the automation of the code generation (or compilation) process, which is an essential component of the technique.

The results are promising:

1. Inspection of the results indicates that in semiautomatic experiments with our synthesis system, most of the connections derived from the specification to the synthesized code are correct, and around half of the lines in the synthesized code can be traced back to at least one line of the specification.
2. In the case of compilation, 75% of the traceability links derived using automatic perturbation involving copying were correct. 20-40% of the lines in the generated assembler could be correctly traced back to the C source program. Perturbations generated by swapping or deleting did not generate any useful traceability links.
3. Small changes in the source often (especially in the GCC examples) induce small changes in the target.

References

[ACPT01] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella. Design-code traceability recovery: selecting the basic linkage properrties. *Science of Computer Programming*, 40:213–234, 2001.

[CE01] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40:171–188, 2001.

[DF03] E. Denney and B. Fischer. Correctness of source-level safety policies. In *Proceedings of FME 2003 (forthcoming)*, 2003.

[FS02] Bernd Fischer and Johann Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 2002. To appear. Preprint available from `http://ase.arc.nasa.gov/`.

[LPPU94] Michael Lowry, Andrew Philpot, Thomas Pressburger, and Ian Underwood. AMPHION: Automatic programming for scientific subroutine libraries. In Zbigniew W. Raś and Maria Zemankova, editors, *8th on Methodologies for Intelligent Systems*, volume 869 of *Lecture Notes in Artificial Intelligence*, pages 326–335, October 1994.

[RG] Julian Richardson and Jeff Green. Traceability through automatic program generation. In *Proceedings of Second International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2003)*.

[WSF02] Michael Whalen, Johann Schumann, and Bernd Fischer. Synthesizing certified code. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *2002: Formal Methods—Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 431–450, Copenhagen, Denmark, July 2002.

[WVS+01] Jon Whittle, Jeffrey Van Baalen, Johann Schumann, Peter Robinson, Thomas Pressburger, John Penix, Phil Oh, Michael Lowry, and Guillaume Brat. Amphion/NAV: Deductive synthesis of state estimation software. In Martin S. Feather and Michael Goedicke, editors, *16th IEEE International Conference on Automated Software Engineering*, pages 395–399, San Diego, CA, November 26–29 2001.