Traceability Through Automatic Program Generation

Julian Richardson[1] and Jeff Green[2]
Automated Software Engineering Group
NASA Ames Research Center, Moffett Field, CA 94035-1000, USA
(1) julianr@email.arc.nasa.gov, RIACS/USRA
(2) jeffreyadamgreen@yahoo.com, QSS

1 Introduction

Program synthesis is a technique for automatically deriving programs from specifications of their behavior. One of the arguments made in favor of program synthesis is that it allows one to trace from the specification to the program.

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.

In this paper, we introduce a novel, 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.

We have partially automated the technique and applied it in an experiment to one of our program synthesis systems, AUTOFILTER, and to the GNU C compiler, GCC. The results are promising:

1. Manual inspection of the results indicates that most of the connections derived from the source (a specification in the case of AUTOFILTER, C source code in the case of GCC) to its generated target (C source code in the case of AUTOFILTER, assembly language code in the case of GCC) are correct.
2. Around half of the lines in the target can be traced to at least one line of the source.
3. Small changes in the source often induce only small changes in the target.

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 [2], AUTOFILTER [5] and before that AMPHION [3]). 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 [4, 1] — 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 [5] — 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 [5]. 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.

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 are changed 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 the nominal target, *"This is not an apple."* Apply separately the perturbations *pomme* $\rightarrow$ *banane*, *pas* $\rightarrow$ *pipe*, and *une* $\rightarrow$ *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 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 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. This greatly reduces the effort needed to employ it, and removes the possibility of inadvertently introducing errors into the synthesis system when it is modified.

3. It does not require detailed, or indeed any knowledge of the internal mechanisms of the synthesis system.

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. There is a single manual part of the process: choosing which perturbations to apply. We discuss this problem below §8.

5 Implementation

Initially, we carried out changes entirely by hand. This indicated that the technique might be interesting, so we decided to automate the process. 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. Note that currently each perturbation may only alter 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 the 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.
  - 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 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. We have not yet tried to formally evaluate the technique.

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 inputs 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 1 shows the source program, and figure 2 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 2 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 Applications

We envisage the following applications of the derived traceability information:

- 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 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.
- Increase learnability. We conjecture that the derived traceability information can make the synthesis process easier to understand for developers unfamiliar with the domain, or with the synthesis mechanism.
- Focus testing. Some parts of the synthesized program vary little or not at all when the specifications are perturbed. Other parts vary greatly. Testing can be focused on the latter.

8 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. Manual inspection of the results indicates that most of the connections derived from the source (a specification in the case of AUTOFILTER, C source code in the case of GCC) to its generated target (C source code in the case of AUTOFILTER, assembly language code in the case of GCC) are correct.
2. Around half of the lines in the target can be traced to at least one line of the source.
3. Small changes in the source often (especially in the GCC example) induce only small changes in the target.

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

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

```
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 2: The annotated assembly language code.

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 perturbations to be applied to the source were chosen manually. They were not systematic: we probably did not find all the ways the source can affect the target, perturbations were probably to some extent targeted (we know what changes make sense and so might lead to meaningful changes in the program). A more systematic methodology would make changes at random in the source. However, we would expect the vast majority of synthesis/compilation attempts on randomly perturbed sources to fail. We could get around this problem by employing a grammar for the source (specification) language to randomly generate perturbations which created grammatically correct perturbed sources.

The technique now needs to be evaluated more thoroughly: we need to apply it to more and different kinds of source and synthesis/compilation systems. To ease automation, only single line changes were made: we should work to remove this limitation because some changes only make sense when made in conjunction with other changes. 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.

References

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

[2] Bernd Fischer and Johann Schumann. AutoBayes: A system for generating data analysis programs from statistical models. 2002. To appear. Preprint available at `http://ase.arc.nasa.gov/people/fischer/`.

[3] 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.

[4] 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.

[5] 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.