TEACHING SPECIFICATION AND VERIFICATION OF EVENT-DRIVEN PROGRAMS USING CLEANROOM SOFTWARE ENGINEERING

Gabriel J. Ferrer
Hendrix College
1600 Washington Avenue
Conway, AR 72032
ferrer@grendel.hendrix.edu

1. INTRODUCTION

Software verification, the process of ensuring that a software artifact meets its specification, is both difficult to perform and difficult to teach. The two principal techniques for software verification are correctness proving and testing. While a correctness proof can demonstrate conclusively that a software artifact meets its specification for all inputs, correctness proving is a difficult, tedious, and error-prone activity [1]. Hence, most software engineers rely in practice upon testing as the primary means of ensuring that a software artifact meets its specification.

Unfortunately, testing only demonstrates that the software artifact meets its specification for the inputs for which it has been tested. Thus, being able to employ proof techniques to demonstrate software correctness is still of interest. Cleanroom Software Engineering [2] is a software engineering methodology that employs semi-formal software specification and theorem-proving techniques in a practical manner to enable more robust verification of software artifacts. Cleanroom has been demonstrated to be very practical in large projects in industry, reducing software defect rates to extremely low levels, typically less than five bugs per thousand lines of source code [3]. (Note that these bug densities are based on counting the total number of bugs found from the first time that *any* code in the system is executed.)

The primary resource available for teaching Cleanroom Software Engineering to undergraduate students is the textbook *Toward Zero-Defect Programming* by Allan Stavely [4]. The book is based upon an undergraduate course taught by Stavely at the New Mexico Institute of Mining and Technology for several years. An important topic that is not addressed by this textbook is the application of Cleanroom techniques to event-driven programs. Many software engineering instructors believe that Cleanroom is not applicable to such programs (e.g. [5]). Therefore, in this paper, a specification technique that the author employs in teaching Cleanroom to undergraduate students is described that illustrates that students can be taught Cleanroom in a manner that enables them to write and analytically verify event-driven computer programs.

The heart of the Cleanroom technique is the development of a functional specification for the program being developed. This is a function that specifies the outputs that ought

to be produced by every possible input. This computational model is quite effective for software that processes an input and then terminates, such as a text formatter or a compiler. However, an event-driven system does not terminate with a single output. In fact, it does not necessarily terminate at all. The key pedagogical idea introduced in this paper is to show how the handling of a single event is analogous to the execution of an entire single-output program for the purpose of specification.

Section 2 gives a definition of the event-driven programming model as well as some motivating examples. Section 3 describes Cleanroom Software Engineering, including a brief overview of its history and track record and a description of the pedagogy for teaching Cleanroom as outlined in Stavely's textbook. Section 4 describes how students can be taught to use Cleanroom for specifying and verifying their event-driven programs. Section 5 discusses the results of applying these techniques in a software engineering course taught by the author in the fall of 2002. Section 6 contains the conclusions and some future directions.

2. EVENT-DRIVEN PROGRAMMING

Traditional imperative computer programs are written with the assumption that the program controls the sequence of steps that occurs at run-time. Under this assumption, a program is a function that deterministically maps its inputs to outputs, terminating when complete.

In contrast, event-driven computer programs do not control the sequence of steps that occurs at run-time; rather, the user controls program flow through the inputs. These inputs, called *events*, may occur asynchronously and are stored in a queue. A simple control loop removes events from the queue one at a time and "handles" each event according to the program instructions. Handling an event can consist of producing an output and/or changing the internal state of the program. These programs are designed to run indefinitely; program execution need not ever terminate.

Mouse-driven graphical user interfaces are perhaps the best-known examples of event-driven programs in current use. Events in such a system include mouse clicks, mouse movements, and keystrokes. Another type of event-driven system of considerable contemporary relevance is the embedded system. Events for embedded systems can include any real-world event relevant to the system. For example, a mobile robot could experience events such as bump sensors indicating a collision or a vision system reporting the recognition of a landmark.

3. CLEANROOM SOFTWARE ENGINEERING

3.1 Background

Cleanroom Software Engineering introduces techniques for specifying, verifying, and

testing software. The techniques for specification and verification as presented by Stavely [4] are the focus of this paper and are described in Section 3.2. This section gives some background and history of Cleanroom.

Prowell et al. [2] state: "Cleanroom software engineering is designed to achieve two critical goals: a manageable development process and no failures in use. The Cleanroom name was borrowed from the hardware cleanrooms of the semiconductor industry, where defect prevention rather than defect removal is pursued through rigorous engineering processes." Harlan Mills, the founder of Cleanroom software engineering, applied two key insights to the creation of the Cleanroom process: first, that a computer program implements a mathematical function, and second, that software testing amounts to sampling from a potentially infinite population of potential software uses. Hence, a program specification is a function that maps inputs (the function's domain) to outputs (the function's range). A correct program can be demonstrated to implement the specification function. Testing serves as a means of statistically verifying that correct code was produced in the development process; the goal is to find all software defects prior to the commencement of testing through demonstrating mathematically that the program is equivalent to the specification function.

Cleanroom has been demonstrated to be of considerable value in large-scale industrial software development, with a very small number of bugs reported in testing. Published estimates of the number of bugs remaining in typical *delivered* code range from 4 to 50 bugs per thousand lines of source code [6] [7]. Projects developed using Cleanroom can achieve lower bug densities measured from the *first* time *any* of the system code is executed. Upon delivery, projects routinely have less than one bug per thousand lines of code. Examples of projects that have used Cleanroom (along with bug densities from the first code execution) include:

- A NASA satellite control system consisting of 40,000 lines of FORTRAN had a bug density of 4.5 per thousand lines of code
- A program to transform COBOL programs to a well-structured form undertaken at IBM consisting of 85,000 lines of PL/1 had a bug density of 3.4 per thousand lines of code
- A device controller for an IBM tape drive consisting of 86,000 lines of C had a bug density of 1.2 per thousand lines of code
- The Ericsson Telecom OS32 operating system consisting of 350,000 lines of C and assembly language had a bug density of 1.0 per thousand lines of code

Again, these bug densities are based on counting the total number of bugs found from the first time that *any* code in the system is executed. Upon delivery, incidences of bugs in Cleanroom-developed projects are rare, and consistently less than one bug per thousand lines of code. For example, in the COBOL Structuring Facility mentioned above, only seven minor bugs were found in the first three years of production use; all of the bugs were easily fixed. These project statistics were obtained from Linger [8].

3.2 Fundamentals of Cleanroom Software Engineering

As mentioned above, a computer program can be specified by a function that maps the set of all possible program inputs to the set of all possible program outputs. Such a function, in Cleanroom terminology, is called an *intended function*. Cleanroom development, as presented in Stavely's textbook [4], begins by specifying an intended function for the entire program. This intended function is then decomposed into a sequence of intended functions that, when composed, are equivalent to the overall intended function of the program. Each intended function is then decomposed into even more intended functions, and ultimately into actual program code. Every line of code in the resulting program is associated with an intended function.

Here is an example of a simple intended function:

```
[ x, y := y, x ]
```

This intended function specifies that x will receive the value of y when the computation terminates, and y will likewise receive the value of x. In other words, this intended function specifies that their values are to be swapped. Assignments in intended functions are concurrent; that is, no sequencing is specified in the computation. Only the output is specified. Here is an example code segment that corresponds to the intended function given above:

```
// [ x, y := y, x]
{
    int temp = y;
    y = x;
    x = temp;
}
```

Intended functions can be conditional, as in the following example:

```
[ x > y -> x, y := y, x
| else -> I ]
```

In this conditional intended function, x and y are swapped only if x > y. Otherwise, the else condition reflects the new system state. The symbol I denotes that no change is made to any variable.

An additional option is to use *specification functions*. Specification functions become important when intended functions become verbose. They provide an important element of specification abstraction. Definitions can be provided for specification functions if they are considered to be ambiguous. A variation of the above intended function using specification functions could be:

```
[ x, y := min (x, y), max (x, y) ]
```

Program verification consists of demonstrating that a given sequence of program instructions produces the same outputs for the given inputs as its corresponding intended function. In the Cleanroom context, the purpose of verification is the discovery of programming errors, as opposed to the production of a written proof of correctness. This

enables the simplification of the verification process, as no formal mathematical artifact need be produced.

The Cleanroom verification process operates as a structured code review. Each type of programming construct has associated with it a set of verification questions. Answering all the questions for a given programming construct affirmatively demonstrates the correctness of that section of code. The programmer presents a verbal justification for why each section of code answers the verification questions affirmatively. This verbal justification may be supplemented with written proofs if other members of the team are not persuaded of the correctness of that section of code. This approach enables a team to avoid being distracted by excessive formality when the correctness of a section of code is reasonably obvious, while ensuring that rigor is employed when truly necessary.

Consider the following intended functions and code. To demonstrate that the code is correct, we must demonstrate that it computes its intended function:

```
// [ x, y := min (x, y), max (x, y) ]
if (x > y) {
    // [ x, y := y, x]
    int temp = x;
    x = y;
    y = temp;
}
```

We begin by verifying the `if` statement. There are two verification questions we must answer:

1. If the condition is true, does doing the body of the `if` statement do the intended function?
2. If the condition is false, does doing nothing do the intended function?

Regarding question 1, if x > y, then `min(x, y) = y` and `max(x, y) = x`. Replacing these specification functions on the right-hand side of the intended function, we find that the intended function for the `if` matches the intended function for the body. Therefore, the answer to question 1 is "yes". Regarding question 2, if x < y, then `min(x, y) = x` and `max(x, y) = y`. Given this condition, the intended function reduces to `[x, y := x, y]`, which is certainly true if no action is taken. Therefore, the answer to question 2 is "yes".

Once the `if` statement has been verified, the body of the if must be verified. By doing some simple substitutions, it is evident that the body does in fact meet its intended function.

4. TEACHING EVENT-DRIVEN PROGRAMMING WITH CLEANROOM

The specification approach described above does not have an immediately obvious

extension to the problem of specifying event-driven programs. All of the examples in Stavely's textbook [4] are of programs that produce an output given a one-time input. Interaction with a user is not modeled in that textbook. Nevertheless, among the Cleanroom success stories mentioned above, the NASA satellite control system, the IBM tape driver, and the Ericsson operating system are all arguably event-driven programs, as they all must handle asynchronous events and they all execute indefinitely. We can conclude from this that Cleanroom can be used for the construction of event-driven systems. What is missing is pedagogical material that unites Stavely's lucid presentation of the Cleanroom method with the idiosyncratic requirements of an event-driven system.

The key idea is to apply the concept of an intended function to the processing of a single event. The input to the program is the event coupled with the program's current state. The output is the program state resulting from handling that event. The intended function with the set of the cross-product of all possible events and all possible system states as its domain and the set of all possible system states as its range is called an *instantaneous intended function.* This intended function can be modeled generically as follows:

```
[ system_state := process_event (event, system_state) ]
```

Verification is reduced to the problem of verifying that the program state after every possible event corresponds to the output of the instantaneous intended function.

To see an example of the application of this concept, consider an event-driven calculator program. Ignoring the issues of memory and operator precedence, its entire system state can be described by the current number in the display, whether a binary operation is incomplete, the identity of any incomplete operation, and the operand of the incomplete operation. Given this context, the instantaneous intended function for the calculator program could be:

```
[ calc_state := process_input (input, calc_state) ]
```

where `input` is either a digit or an operation, and `calc_state` is a triple consisting of: `(digit_sequence, pending_operator, last_number)`. The purpose of `pending_operator` and `last_number` is to record the state of a binary calculation that is in progress. If no such calculation is in progress, both `pending_operator` and `last_number` will have the value `null`. If a calculation is in progress, `pending_operator` stores the specific binary operator and `last_number` stores the previously entered operand. The specification function `process_input` is defined as follows:

```
process_input (digit, (0, any, any)) = (digit, any, any)
process_input   (digit,   (digit_sequence,   any,   any))   =
          (digit_sequence + digit, any, any)
process_input (unary_operator, (digit_sequence, any, any)) =
          (unary_operator (digit_sequence), any, any)
```

These first three clauses ensure that when a digit is entered into the calculator, it is concatenated to the existing number, unless the existing number is zero, which is then

replaced. Applying a unary operator to a number replaces the number with the result of applying that operation. The set of unary operators includes negation, common and natural logarithms, square root, and the basic trigonometric functions.

```
process_input (binary_operator, (digit_sequence, null, null)) =
          (0, binary_operator, digit_sequence)
process_input (=, (digit_seq_1, binary_op, digit_seq_2)) =
          (binary_op (digit_seq_2, digit_seq_1), null, null)
```

The two clauses above regard a single binary calculation. If no binary operation is pending, clicking a button for a binary operation transfers the current number from the display to the calculator's memory. If a binary operation is pending, clicking the equal sign causes the result of applying that operation to be displayed. The set of binary operators includes addition, subtraction, multiplication, and division.

```
process_input(binary_op_1,(digit_seq_1,binary_op_2,digit_seq_2) =
          (0, binary_op_1, binary_op_2 (digit_seq_2,digit_seq_1))
```

The above clause regards the situation in which a binary operation is concluded with the commencement of a second binary operation. In that case, the result of the calculation is stored in the calculator's memory to be used in conjunction with the second operation.

The above example is somewhat simplified, but it should be sufficient to demonstrate the utility of the instantaneous intended function. The specification maps inputs to outputs in such a way that a programmer may verify whether his code matches the specification according to the Cleanroom methodology.

In the context of Cleanroom Software Engineering, the purpose of testing is primarily to check the quality of the resulting software rather than finding bugs. Testing an event-driven program in the Cleanroom style was a straightforward extension of grammar-based test generation as described in Chapter 10 of Stavely's textbook [4]. Students constructed grammars to represent sequences of events generated by a user. Using student-constructed test case generators, they generated scripts of event sequences from those grammars. They then performed each event in sequence as specified in each script.

5. CLASSROOM EXPERIENCE

The author’s software engineering students in the Fall semester of 2002 implemented an event-driven interpreter for the Logo programming language, complete with turtle graphics. Cleanroom software engineering and the instantaneous intended function pattern were used in the software development process. The class consisted of six juniors and two seniors, all of whom have had at least three prior programming courses. Defect levels in student code are similar to what has been reported previously when teaching Cleanroom [9]. One student team discovered four defects in 1328 lines of code from the

first execution in testing. The other student team discovered only one defect in 1436 lines of code.

Here is the top-level instantaneous intended function devised for this project by one of the student teams. The system state is represented by a triple.

```
[ (text_editor_state, command_line_state, disk_state) :=
    process_input (event, text_editor_state, command_line_state,
                   disk_state) ]
```

The top-level definition of the process_input specification function is:

```
process_input  (event,  text_editor_state,  command_line_state,
                   disk_state) =
   (process_text_editor_state (event, text_editor_state),
    process_command_line_state (event, command_line_state),
    process_disk_state (event, disk_state))
```

This student team produced a six-page specification document that further detailed the specification functions for the different state elements of the system.

Students commented that the activity of writing the instantaneous intended function was very helpful in determining what code needed to be written. The author was told that the coding process itself was a very straightforward task once the specification was completed. The primary source of the few run-time bugs that did occur was student unfamiliarity with the behavior of the GUI toolkit that they were using.

6. CONCLUSIONS

Our classroom experience has demonstrated that the concept of the instantaneous intended function is a helpful pedagogical technique for teaching students how to develop software using Cleanroom Software Engineering for the development of event-driven software. Students successfully used the instantaneous intended function in the context of Cleanroom-style verification. The rate of student run-time programming errors was extremely low and consistent with previously reported results regarding the efficacy of teaching Cleanroom using the educational materials provided by Stavely's textbook.

Further development of course materials for applying Cleanroom techniques to event-driven programming is in the planning stages. One potential topic is an investigation of a potential relationship between the instantaneous intended function and object-oriented analysis, with a particular emphasis on the use-case concept from UML [10]. This could be helpful in validating that an instantaneous intended function describes a software system that matches the needs of the customer for whom the software is being developed. An additional topic under consideration involves the implications of multithreaded applications for the writing of intended functions in an event-driven context. In particular, we are interested in exploring how thread synchronization and thread preemption might be expressed with the semantics of intended functions.

REFERENCES

[1] P. Jalote. *An Integrated Approach to Software Engineering.* Springer-Verlag, 1991.

[2] S. J. Prowell, C. J. Trammell, R. C. Linger, J. H. Poore. *Cleanroom Software Engineering: Technology and Process.* Addison-Wesley, 1999.

[3] P. A. Hausler, R. C. Linger, and C. J. Trammell. "Adopting Cleanroom software engineering with a phased approach." *IBM Systems Journal* 32:2, pp. 232-251, 1993.

[4] A. Stavely. *Toward Zero-Defect Programming.* Addison-Wesley, 1999.

[5] T. Wahls. "Course Notes for CS416/516." http://mathcs.hood.edu/~wahls/416/process.html, 2002.

[6] C. B. Jones. *Programming Productivity.* McGraw-Hill, 1986.

[7] T. DeMarco. *Controlling Software Projects: Management Measurement and Estimation.* Prentice-Hall, 1982.

[8] R. C. Linger. "Cleanroom process model." *IEEE Software* 11:2 pp. 50-58, March 1994.

[9] A. Stavely. "High-quality software through semiformal specification and verification." In *Twelfth Conference on Software Engineering Education and Training.* http://www.nmt.edu/~al/cseet-paper.html, March 22-24, 1999.

[10] M. Fowler and K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley, 1999.