Formal Safety Analysis of the Control Program for a Radiation Therapy Machine

Jonathan Jacky
Radiation Oncology, Box 356043, University of Washington, Seattle, Washington 98195-6043, USA

Introduction

A central problem in software development is to assure that a complex program actually meets its requirements. This problem is most urgent in applications such as the control of radiation therapy machines, where execution of the program can have irreversible human consequences.

The problem is difficult for at least two reasons. First, there is the sheer mass of detail that must be handled correctly: the written requirements for our machine's control program comprise hundreds of pages of prose, and the printed program listing comprises hundreds of pages of programming language statements (code)[1]. The second difficulty arises because the two descriptions are so different. The requirements are expressed in English prose organized to be meaningful to the machine's users and maintainers, while the program is expressed in a machine-readable code organized to be efficiently executed by the computer. There is no obvious (simple, one-to-one) correspondence between the two. Yet we must somehow derive the code from the requirements. This is the *design* problem. Then we are obligated to show that the code satisfies the requirements — not approximately, but exactly, with no errors or omissions and no extra (undocumented) behavior. In the course of this effort we should discover and correct all errors that might have serious consequences. This is the *assurance* problem. How should we solve these two problems?

The usual approach to solving both problems is to create one or more intermediate descriptions between the requirements and the code. This description (or collection of descriptions) is usually called the *design.* It helps to solve both problems by replacing the huge leap from requirements to code with smaller steps, which should be easier to create (design) and justify (assure).

The choice of design method and notation is an active area of research and controversy. For this project we chose to use a *formal method* and express the design in a *formal notation.* This means that the design is a collection of mathematical formulas. In the literature this is usually called a *formal specification* (even though it often serves as the design).

We expected two benefits. First, we hoped to create a design document that was more concise and more clearly structured than we could achieve with prose or diagrams, so it would be more convenient to use as a guide for coding, test planning, and assurance activities (reviews). Second, we hoped to apply powerful assurance techniques that become possible when a formal specification is available. A formal specification is a kind of mathematical model, so it is sometimes possible to infer or calculate conclusive answers to questions about proposed system behaviors.

In particular, we hoped to demonstrate formally that the detailed design satisfies its safety requirements. The central safety requirement is that the beam can only turn on when the actual setup of the machine conforms to the prescription selected by the operator[2]. Checking this requirement might reveal errors that escape detection during testing and reviews. To make the safety analysis possible, the formal specification must explicitly describe the safe states. This information is not present in the program and cannot be inferred from the program.

Our innovation in this project, which we believe is a novel and significant scientific contribution, was to create a formal specification of a radiation therapy machine control program in sufficient detail that it could be implemented and also could be analyzed to determine whether it satisfies its safety requirements. We coded the program from the formal specification and it is now in use treating patients. We also experimented with formal assurance methods, but we relied primarily on traditional assurance methods: testing and code reviews.

Material and methods

We developed a new control program for the isocentric treatment unit at the Clinical Neutron Therapy System (CNTS) at the University of Washington Medical Center in Seattle. Except for particle type, this machine is similar to a modern therapy linac with a leaf collimator. In July 1999 our new program replaced the manufacturer's original program which had been in use since the machine was installed in 1984 [1]. There was no re-use of design or code from the earlier program.

Writing the requirements

The requirements express the properties that the system must have to be acceptable to its users, expressed in terms that they can understand and critique. The authors and reviewers include the physicist who defined the physical and clinical requirements for the original machine, engineers who installed and maintain the machine, and clinical users of the machine. Writing the requirements was a major portion of the whole project effort, not just a preliminary. It was an indispensable prerequisite to the formal specification.

[1] Most project documents are available at `http://www.radonc.washington.edu/physics/cnts/`

[2] Other safety requirements are handled by hardware. Delegation of functions to hardware or software follows a hazard analysis not reported here.

Writing the formal specification

The formal specification is a collection of mathematical formulas that express constraints on the values of a set of *state variables.*

The set of state variables must represent all pertinent features of the treatment machinery and the treatment session. Most obviously, there are state variables for all the inputs and outputs of the control hardware: the commanded and achieved values for all the leaves, etc. Other state variables represent the status of operations in progress (timers are one example). Still more state variables represent the prescribed settings (needed to check the safety requirement) and the patient and field identification used to select the prescribed settings from the prescription database. We identified 410 scalar state variables.

We wrote formulas to express the constraints that must hold among the values of the state variables at all times. These formulas are called *invariants.* All safety requirements can be expressed as invariants. The formulas are similar to the boolean expressions available in most programming languages: they evaluate to *true* in situations that satisfy the constraints and *false* otherwise. The English paraphrase of one invariant says: "If the beam is on, all leaves are within tolerance of their prescribed values". A condition that causes an invariant to evaluate to *false* violates a safety requirement. For example if any leaf is not within tolerance of its prescribed value when the beam is on, this invariant is violated[3].

The invariants express precisely what it means for the program to be in a safe state. Most of the value added to the project by the formal specification inheres in the invariants because they express information that is not present in the program and cannot be inferred from the program. The program code expresses a great many state transitions. With great effort, we might be able to analyze the code to determine the set of reachable states. But this cannot tell us if the reachable states are safe. The program might be able to reach an unsafe state, due to a designer's oversight or a programming error. In fact, the safety analysis is actually nothing more than checking that the set of reachable states is a subset of the set of safe states. For this we need an explicit description of the safe states which is independent of the program, and that is what the formal specification provides.

In addition to the invariants, we also wrote formulas to express the state transitions. Unlike the invariants, these formulas correspond closely to the program code. However they are written in the same mathematical notation as the invariants, and they are more concise than code because they only model its effects, not its details (for example there are no constructs corresponding to loops in the formal specification). Each operation is described by a formula called a *precondition* which describes the states where the operation can begin and another called a *postcondition* which describes the state after the operation completes (the postcondition can be a function of the precondition). We described 105 state transitions.

[3]The formula paraphrased *if p then q* is *false* when *p* is *true* and *q* is *false*, and is *true* in all other situations.

We wrote the formal specification in Z [2], a machine-readable notation for logic, set theory and arithmetic. Z defines many constructs that make it convenient to work with large numbers of state variables. We wrote 2103 lines of Z.

Analysing the formal specification

A formal specification makes it possible to perform systematic analyses that can reveal design errors. Otherwise, these errors might not be detected until testing, and perhaps not even then. Some of these analyses can be performed automatically by a computer program.

We checked for syntax and type errors, using a tool similar to a programming language compiler. This reveals many of the clerical errors that can creep into large documents (such as different variables with the same name).

We also checked for *domain errors,* using an *automated theorem prover* [3]. This can reveal errors somewhat like array-out-of-bounds errors in programming languages, where a function is applied outside its domain.

We checked portions of the formal specification for *completeness* (there is a specified response to all possible inputs in all possible states) and *determinism* (there is just one response for each input in each state). It should be possible to automate much of this, but instead we inspected pertinent excerpts of the formal specification presented in a tabular format.

Our most important analysis was to check that the detailed design of the program code (the state transitions) satisfies its safety requirements (the invariants). To demonstrate safety we must show that every reachable state is safe. First, show that the initial states are safe. Then, for each state transition, show that if its starting state is safe, then its ending state is safe also. That's all. It is not necessary to consider the practically infinite collection of all possible program executions (we need not explicitly construct the entire (huge!) sets of safe states or reachable states). We need only consider the initial state (trivial) and then individually analyze each of our 105 state transitions (tedious but quite feasible). We check that each transition does not violate any invariants. It is not necessary to explicitly consider all possible values of the state variables because the transitions and the invariants are both represented symbolically (with variables instead of explicit values). We need only show that the formula that describes the transition implies the formula that expresses the invariants. This is usually a straightforward exercise in the simplification of logical expressions. Again, it should be possible to automate much of this, but we used inspection (by eye) and a few pencil-and-paper calculations.

We also analyzed some of our design's tasking and timing properties with an exhaustive simulation technique called *model checking* [4]. We expressed portions of our design in a special programming language used by the checker. Then we expressed certain requirements in another formal notation called CTL that includes temporal operators such as *until* and *eventually* so it can check progress properties, not just invariance. The checker tool constructs the (very large but finite) set of states reachable by the (simplified) program, performing a simulation of all possible program executions. The tool checks that each simulated execution path satisfies the requirements expressed in CTL. If it does not, it prints the entire execution path that contains the error. This technique depends on constructing a much-simplified version of the program that nevertheless captures the requirements of interest.

Writing and testing the program

The formal specification not only supported the safety analysis, it also served as the detailed design that guided coding and test planning. We used no other design notation.

Coding a program from a formal specification as detailed as ours is a straightforward exercise. State variables in the specification are implemented by program variables (of course there are many additional program variables: loop indices, etc.). State transitions are implemented by functions. Usually we can implement each Z construct in less than a page of C code so it is easy to confirm that the code correctly achieves the intent of its specification. We performed pencil-and-paper formal derivations of a few pages of code, but we derived most of the code by intuition and verified it by inspection, using methods recommended in a textbook [2]. The program is 15,786 lines of C code.

The formal analyses described in the preceding section were applied to the formal specification, not the code. It was intended to reveal design errors, and cannot reveal coding errors. Our development method should ensure that, at worst, the code only contains trivial logic errors. We used code inspections and testing to detect these errors.

Testing revealed 213 coding errors during development before the program was considered complete and 13 errors during testing of the completed program.

Results and discussion

The new program began clinical operation in July 1999. In the first four months of use, six errors were revealed. All six involved incorrect warning messages or incorrect record-keeping by the program. At this writing (December 1999), no more errors have appeared. The program has never set up a field incorrectly or enabled the beam to turn on or remain on when the machine was not set up correctly.

Conclusion

There is really only one way to show that a program possesses any invariance property (such as safety): show that the set of reachable program states is a subset of the set of acceptable (safe) program states. There are only two techniques that can show this conclusively: theorem proving and model checking. Both techniques require an explicit description of the acceptable states which is independent of the program.

Except for testing, all other effective assurance methods can be seen as more or less informal approximations to theorem proving or model checking. Our contribution in this project was to raise the level of formality, thereby (we believe) improving the effectiveness of the analysis. We have shown that it is feasible to create and use formal specifications for radiation therapy machine controls. However, our manual analysis effort was tedious and fallible. The obvious next step is to investigate the effectiveness of automated theorem provers and model checkers.

Acknowledgements

Thanks to Rob Emery, Ira Kalet, Dave Reid, Ruedi Risler, and Peter Wootton for work on requirements, testing, installation and use, Michael Patrick and Jonathan Unger for work on the formal specification and code, Dave Reid also for work on code, and Mark Saaltink for help with Z/EVES.

References

[1] Ira J. Kalet, Jonathan P. Jacky, Ruedi Risler, Solveig Rohlin, and Peter Wootton. Integration of radiotherapy planning systems and radiotherapy treatment equipment: 11 years experience. *International Journal of Radiation Oncology, Biology and Physics*, 38(1):213–221, 1997.

[2] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods.* Cambridge University Press, 1997.

[3] Mark Saaltink. The Z/EVES system. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation*, pages 72 – 85. Tenth International Conference of Z Users, Springer-Verlag, 1997. Lecture Notes in Computer Science 1212.

[4] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J. W. De Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, pages 124–175, Noordwijkerhout, The Netherlands, 1993. REX School/Symposium, Springer-Verlag. Lecture Notes in Computer Science, vol. 803.