Automatic Generation of Provably Correct Software from Formal Specification Languages

Senior Capstone Experience

Robert Dickerson

May, 2006

Background

It is widely recognized that software errors are the source of substantial social and economic costs. A 2002 study conducted by the United States Department of Commerce's National Institute of Standards and Technology (NIST) estimated that software errors cost the U.S. economy up to $59.5 billion annually, or about 0.6 percent of the nation's GDP.[1] The same study estimates that about $22.2 billion of this cost could be eliminated by using improved development practices that detect software flaws earlier.[2] Indeed, the cost of rectifying errors that are not detected before the software implementation phase are dramatically higher than those that are detected in the design phase or earlier.[3]

For some applications, the cost of a software fault can very large. An error in an government-used communications program can jeopardize national security; an error in an air traffic control system can cost lives. In these situations, rigorous testing might not be enough to ensure a minimal amount of programming error. Edsger Dijkstra is famously quoted as saying, "program testing can be used to show the presence of bugs, but never to show their absence."[4] To demonstrate the absence of error, a more rigorous program verification method must be developed.

Working with Praxis Critical Systems Limited, John Barnes has described an idea called "correctness by construction," which "encourages the development of programs in an orderly manner with the aim that the program should be correct by virtue of the techniques used in its construction."[5] Praxis attempts to achieve this model of program construction by introducing mathematical proofs of correctness at the specification level, followed by static condition-verification methods at the implementation level. While Praxis does not claim to develop completely error-free code, the methodology employed by the company yields an average error rate of less than one flaw per 10,000 lines of code; somewhere between 50 and 1,000 times better than the estimated industry average.[6]

However, before any method to mathematically verify or construct a correct program can be used, a program must be made expressible as a mathematical object. In a 1984 paper, C.A.R. Hoare and F. K. Hanna make the argument that "[a] computer program is identified with the strongest predicate describing every relevant observation that can be made of the behavior of a computer executing that program."[7] Hoare and Hanna then argue that "a specification is a predicate describing all permitted observations of a program," and so "[a] program P meets its specification S iff $\vDash P \Rightarrow S$."[8] By defining a program and its specification in terms of logical predicates, each object may be analyzed with mathematical techniques.

[1] RTI 2002.
[2] RTI 2002.
[3] Schach 2005, p. 14.
[4] Backhouse 1986, p. 3.
[5] Barnes 2003, p. 5.
[6] Barnes 2003.
[7] Hanna 1984, p. 475.
[8] Hanna 1984, p. 475.

Of course, once specifications and pieces of software have been established abstractly as predicates, it is natural to look for specific ways of representing each artifact so that their particular conditions and properties may be mathematically verified. For constructing program specifications, a number of special languages have been developed, including Z[9], B[10], and the Vienna Development Method (VDM)[11]. Most of these languages use Zermelo-Fraenkel set theory together with standard First Order Logic to express the conditions a piece of software must meet in a mathematically rigorous manner.

Methods for representing software in a manner conducive to mathematical analysis vary according to the type of programming language under consideration. According to Hoare and Hanna, every programming language "is a subset of logical and mathematical notations, which is so restricted that products described in the language can be automatically implemented in a computer,"[12] which seems to imply that every program may be analyzed mathematically. In practice, however, it is difficult to find a general proof technique that works across every language, and so most programs that are to be subjected to formal analysis are written in a language designed to make such analysis more straightforward.

The Spark-Ada language contains syntax for expressing functional pre- and post-conditions, as well as specifying which subprograms may change or use given variables. These statements about the program logic are then verified before compilation to ensure that the resulting software will have the properties specified in the source code. In general, procedural programs can be analyzed with pre- and post-condition verification. A technique described by Backhouse proves properties of code segments by applying pre- and post-condition analysis on a statement-by-statement level.[13]

Programs written for logic programming languages (such as Prolog) may be analyzed in a fashion similar to specification documents, since many of these programs are essentially program specifications anyway. R. Kowalski argues effectively in a 1984 paper titled *The Relation Between Logic Programming and Logic Specification* that the differentiating factor between a specification and a program is the level of detail given in how each procedure is to be carried out.[14] Since logic programming languages use techniques like resolution to determine the details of computing relations given in program code, Kowalski argues that the programs written in these languages are vague enough to be considered executable specification documents.

As Kowalski points out, having an executable specification document blurs the line between program design and implementation, since "[t]he only criterion that can be used to discriminate between [logic programs and complete logic

[9] Bowan 2006.
[10] Bowan 2005.
[11] Fitzgerald 2005.
[12] Hanna 1984, p. 1.
[13] Backhouse 1986, p. 86.
[14] Kowalski 1984, p. 1.

specifications] seems to be relative efficiency. . . "[15] Being able to directly execute a program specification in either a logic-based or procedural language certainly seems beneficial, if for no other reason then providing a tool for prototyping and testing at the specification and design phases of software development.

The Z Specification Language

As previously mentioned, a mathematically rigorous way of representing specification documents must be developed before formal analysis of any document may be carried out. One such method of representing specifications is the Z programming language, developed by the Programming Research Group at the Oxford University Computing Laboratory (OUCL) in the late 1970's.[16] This language is based on first order logic and Zermelo-Fraenkel set theory, and thus allows for standard logical operators and both existential and universal quantification when describing required program behavior. Z also supports the creation of types, which are formed using set theory with first order set comprehension. These features makes Z ideal for model oriented specifications.

A Z document consists of a collection of schemata, which describe the attributes of program structures, and operations, which describe the conditions that must be satisfied by subprograms. Schemata and operation conditions are described in typed first order logic, with operation conditions using an apostrophe (') to denote "final" values of variables. For example, a simple counter object might contain a schema defining ctr to be an integer value with the invariant condition $0 \leq ctr \leq max$, as well as an increment operation with the conditions $ctr < max$ and $ctr' = ctr + 1$.

By using the invariant conditions specified by each section of the Z document, it is possible to prove that invariants specified elsewhere in the document will not be broken. Tools exist to perform syntax and type checking for Z documents[17], and automated theorem provers like ProofPower can be used to generate rigorous proofs of program conditions. However, it is important to note that Z documents may only be check for consistency in relation to themselves; if a semantically incorrect behavior is specified in a Z document, but is syntactically correct and does not violate invariant conditions elsewhere in the document, then it will not be flagged by a Z type checker or proof utility. For example, if a Z operation specifies that a counter increment procedure meet the conditions:

$$ctr > 0 \text{ and } ctr' = ctr - 1$$

instead of:

$$ctr < max \text{ and } ctr' = ctr + 1$$

then a Z verification tool will not raise an error, since the statement is formatted correctly and does not violate any invariant conditions placed on the *ctr* variable.

[15] Kowalski 1984, p. 357.
[16] Bowan 2006.
[17] Bowan 2006.

`--# global`	Permits access to a global variable
`--# derives`	Defines interdependencies between imports and exports of subprograms
`--# main_program`	Indicates the main subprogram
`--# own`	Announces variables declared within packages which thus have state
`--# initializes`	Indicates that the given own variables are initialized before the main subprogram is entered
`--# inherit`	Permits access to entities in other packages
`--# hide`	Identifies text that is not to be examined

Table 1: Spark Core Annotations. Taken from High Integrity Software by John Barnes[20]

The Spark-Ada Method

After a formal specification document has been created and verified, a program must be written that meets the requirements set by the specification. As previously mentioned, a company called Praxis Critical Systems Limited has developed a language that facilitates the automation of proofs that code segments meet their specification requirements. The Spark-Ada method uses static verification utilities to prove the correctness of programs written in the Spark-Ada language. This language is a subset of Ada together with a set of annotations that express verifiable conditions to Spark utilities, but are recognized as comments by an Ada compiler; thus, Spark-Ada programs may be compiled with a standard Ada compiler.[18]

A group of core annotations in the Spark-Ada language provide ways for a programmer to specify the conditions a code segment is required to meet. These core annotations are described in Table 1.

The annotations `--# post` and `--# pre` may also be used to explicitly declare pre- and post-conditions of a code segment. To express the initial (pre-execution of the subprogram) value of a variable, Spark notation affixes a tilde to the end of the variable name. So, a subprogram that increments a counter variable might look like:

```
procedure Increment()
--# global in out ctr;
--# pre ctr < max;
--# post ctr = ctr~ + 1;
is
begin
```

[18]Of course, since most compilers have not been proved correct themselves, all executable code they generate cannot be considered rigorously "proved correct." Barnes notes that this is the case, and offers as a consolation the fact that a compiler is heavily used and has been in existence for a long while, and so is likely to contain very few errors. However, it is important to note that, while a Spark-Ada program may be proved correct, the executable byte code that is generated by a compiler from such a program is not.

```
     ctr := ctr + 1;
end Increment;
```

After Spark-Ada code is written, a tool called the Spark Examiner is used to "check conformance of the code to the rules of the kernel language" and "check consistency between the code and the embedded annotations by control, data and information flow analysis."[21] This is done by automatically producing verification conditions which may be verified by an automated theorem checker.[22] These tools allow properties and conditions of Spark-Ada programs to be proved statically before program compilation.

Proving Properties of Code Segments

To accomplish these proofs of code segment properties, the Spark Examiner makes use of pre- and post-condition analysis. Most programmers are familiar with subprogram level pre- and post-conditions; a postcondition of a subprogram is a condition that must hold after the subprogram has finished running, provided that all preconditions of the subprogram were true just before the subprogram began running. For example, a procedure that removes an item from an array might have "item x is in the array" as a precondition and "item x is not in the array" as a postcondition.

Pre- and post-conditions at the subprogram level are useful for reasoning about overall program behavior, but demonstrating that a subprogram meets its postconditions given that its preconditions are satisfied requires analysis at the single statement level. At this level, pre- and post-conditions are also a viable tool for reasoning about behavior of the program. For example, given the precondition "$a = b$ and $c = d$", the statement $b := c$ will have the postcondition "$a = c$ and $c = d$." This paper will adopt notation used by Backhouse, where $\{P\}S\{Q\}$ represents the statement "given precondition P, statement S satisfies postcondition Q."[23]

At this point, it is useful to consider C. A. R. Hoare's rule of sequential composition,[24] which states that:

$$\{P\}S_1\{R\} \wedge \{R\}S_2\{Q\} \Rightarrow \{P\}S_1; S_2\{Q\}$$

This, in effect, allows a group of statements to be chained together, with the precondition of the grouping being the precondition of the first statement, and the postcondition of the grouping being the postcondition of the final statement. So, a subprogram $S_0; S_1; \ldots; S_n;$ can be said to meet a postcondition Q given a precondition P iff there exist conditions $\{C_0, C_1, \ldots, C_n\}$ such that $\{C_i\}S_i\{C_i + 1\}$ and $C_0 = P$ and $C_n = Q$.

To prove that a subprogram $S_0; S_1; \ldots; S_n;$ meets the necessary postcondition given a precondition, the Spark Examiner searches for a sequence of

[21] Barnes 2003, p. 14.
[22] Barnes 2003, p. 15.
[23] Backhouse 1986, p. 86.
[24] Backhouse 1986, p. 95.

satisfactory statement level conditions $C_0, C_1, \ldots, C_n$ with a technique called hoisting.[25] This technique makes use of the weakest precondition concept, which Backhouse has defined as follows:

> Definition. If Q is a predicate and S a statement, then the weakest precondition of S with respect to Q, denoted $wp(S, Q)$, is a predicate describing the set of all initial states such that the execution of S begun in any one of the states is guaranteed to terminate in a state satisfying Q.[26]

The hoisting technique begins with the subprogram's postcondition Q and the last statement in the subprogram, S_n. It then calculates $wp(S_n, Q)$ and uses this new condition together with statement S_{n-1} to find a new precondition, and so on until the weakest precondition of the first statement (S_1) is found. If this condition implies the precondition for the subprogram, then the subprogram has been successfully proved to meet its postcondition with a given precondition.

Automating the Specification-to-Program Transition

As previously mentioned, programs written in logic programming languages (such as Prolog) have been argued to be a type of executable specification document, since running a program in such a language amounts to backward chaining to find a suitable solution to the posed question. Finding code segments in procedural languages (such as Spark-Ada) that satisfy a formal specification document may also be accomplished with a backwards chaining algorithm, making use of the weakest precondition concept in a way similar to hoisting.

In the hoisting technique, a weakest precondition is generated from the desired postcondition and the statement currently being examined. If, however, the desired postcondition were all that were known, a list of possible statements that would generate a non-trivial weakest precondition could be generated. Then, a search tree could be constructed as follows:

Let P be the desired precondition of a subprogram and Q be the desired postcondition. Each node of the tree consists of an ordered pair $(S, wp(S, C_p))$, where S is a statement in the desired language and C_p is the second element in the parent node's ordered pair. Define the root node of the tree to be (δ, Q), where δ is the empty string. To expand the tree, add as children to each leaf node (S, C_t) the elements of the set $\{(s_i, wp(s_i, C_t) | s_i \text{ a statement in the language and } wp(s_i, C_t) \neq C_t\}$.[27]

To generate a subprogram that provably meets desired pre- and post-conditions, continuously expand the tree until a child node is found that is implied by the

[25] Barnes 2003, p. 240.

[26] Backhouse 1986, p. 88.

[27] If this set is infinite, add one element as a child each iteration of the search. The resulting search algorithm will remain complete and correct, but could take much more time to run. (However, the search algorithm runs in non-polynomial time to begin with).

precondition of the subprogram. Call this node R, and let

$$\{(S_0, C_0 = wp(S_0, C_1)), (S_1, C_1 = wp(S_1, C_2)), \ldots, (S_n, C_n = wp(S_n, Q)), (\delta, Q)\}$$

be the set of nodes in the path that leads from R to the root. Then, the subprogram $S_0; S_1; \ldots; S_n$; must meet Q given P, since $P \Rightarrow C_0$ and

$$\{C_0\}S_0\{C_1\} \wedge \{C_1\}S_1\{C_2\} \wedge \cdots \wedge \{C_n\}S_n\{Q\}$$

which implies

$$\{P\}S_0; S_1; \ldots; S_n\{Q\}$$

by the rule of sequential composition.

Using this type of search algorithm, a subprogram in a procedural language can be generated from the subprogram's desired preconditions and postconditions alone.

Summary

Currently, high integrity software companies like Praxis develop software by writing specifications in formal first order languages (such as Z) and then translating the specifications into languages (such as Spark-Ada) that are conducive to static proof techniques. It is interesting to consider automating the transition from the formal specification document to provably correct executable code, especially for prototyping and testing specification documents.

Logic programming languages, in a sense, already do this via backward chaining (usually using the resolution principle). Since subprogram conditions expressed in formal specification languages are readily translated into formal subprogram preconditions and postconditions, it is easy to establish a starting state and goal state for a search algorithm to generate a subprogram along with a proof of its correctness. Using the idea of a weakest precondition, such an algorithm can be constructed by chaining between the starting and goal states.

While this type of search algorithm is not efficient enough to be very practical for generating large amounts of code, it is interesting to consider optimizations and parallelization that could make the automatic generation of software from specification documents a useful software development tool.

References

Backhouse, Roland C. (1986). *Program Construction and Verification*. Englewood Cliffs: Prentice-Hall International.

Barnes, John (2003). *High Integrity Software: The SPARK Approach to Safety and Security*. Edinburgh Gate: Pearson Education Limited.

Bowan, Johnathan (2005). *The B-Method*. URL: `http://vl.fmnet.info/b`.

— (2006). *Z Frequently Asked Questions (FAQ)*. URL: `ftp://ftp.comlab.ox.ac.uk/pub/Zforum/faq.txt`.

Fitzgerald, John (2005). *Information on VDM and VDM++*. URL: `http://www.csr.ncl.ac.uk/vdm`.

Hanna, C. A. R. Hoare F. K. (1984). "Programs are Predicates". In: *Philosophical Transactions of the Royal Society of London*. A, Mathematical and Physical Sciences, pp. 475–489.

Kowalski, R. (1984). "The Relation Between Logic Programming and Logic Specification". In: *Philosophical Transactions of the Royal Society of London*. A, Mathematical and Physical Sciences, pp. 345–361.

RTI (2002). *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Tech. rep. NIST. URL: `http://www.nist.gov/director/prog-ofc/report02-3.pdf`.

Schach, Stephen R. (2005). *Object Oriented and Classical Software Engineering*. New York: McGraw Hill.