Automating Software Fault Tolerance

Christian Wild*
Old Dominion University, Norfolk, Virginia

It is conjectured that the production of software for ultrareliable computing systems such as required by space station, fly-by-wire aircraft and nuclear power plants will require a high degree of automation as well as fault tolerance. In this paper, some of the results of a study of the relationship between automatic programming techniques, formal specifications, and fault tolerant computing systems are presented. The two aspects of fault discussed in this paper are the derivation of assertions for detecting errors in operational computing systems and the generation of diverse designs capable of continued service in the presence of faults. An approach to the automatic generation of assertions for run-time monitoring of computing systems and the generation of test cases from formal specifications is outlined.

Introduction

THE primary objective of this research is the implementation and operation of autonomous and survivable computing systems. An autonomous computing system will perform its mission without extensive and continuous monitoring by humans. A survivable system can continue to meet critical mission objectives even in the face of hardware, software, and environmental faults. We believe that the production of such systems must be highly automated in order to achieve these goals. This paper reports the results of a feasibility study of the use of automatic programming technology and formal specification techniques in the production of ultrareliable computing systems. Although the emphasis of this study is on software design fault tolerance, many of the techniques are also applicable to hardware and environmental faults as well.

These are two divergent schools of thought on the implementation of reliable software. The fault avoidance school attempts to eliminate software design faults through careful design methodology and formal proofs. The fault tolerance school takes the approach that correct code may be impossible or prohibitively expensive to implement. Faults that may occur are tolerated through the provision of redundancy. It is our feeling that some degree of fault tolerance will be present in autonomous survivable systems. Automatic programming technology may be able to address some of the present shortcomings of fault tolerant software, including errors in specification, correlated failure modes of "independent" versions, acceptance test generation, global behavior monitoring, and reconfiguration on catastrophic failures. The two aspects of fault tolerance addressed in this paper are error detection and the generation of diverse designs capable of continued service in the presence of faults. The next section presents background on software fault tolerance. A more detailed discussion of these issues can be found in Ref. 1.

Software Fault Tolerance

The provision of fault tolerance requires the consideration of the issues of error detection, damage assessment, error recovery, and continued service.[2] All fault tolerant techniques must start with error detection,[3] which requires some form of redundancy to signal the presence of the erroneous condition. Damage assessment refers to the process of determining the extent of the damage caused by the fault. Because of the difficulty of making an accurate assessment of damage at run time, the extent of potential damage is usually predetermined by an analysis of the structure of the program. Error recovery is the process by which the damage is fixed and the system is brought back to a consistent state. This can be accomplished either by returning to a consistent previous state (backward recovery) or by correcting the current inconsistent state (forward recovery). For software design faults, continued service implies using an alternate design to achieve the intended goal, called "design diversity" by Avizienis.[4]

Several software fault tolerant methodologies have been proposed.[5] Two well-known techniques are multiversion programming[6] and recovery blocks.[7] Both of these techniques rely on the development of redundant versions of the software. In multiversion programming, the results of the redundant versions are compared in order to arrive at a consensus, a "correct" answer. For recovery blocks, the results from one version must pass an acceptance test. If the acceptance test fails, then an alternate version is executed and its results are subjected to the acceptance test. This process can continue until all the redundant versions are tried.

These techniques have seen only limited use in actual systems, and there remain several fundamental issues to be resolved. For example, it is assumed that the specification contains no errors (a shortcoming also shared by program verification). There is the important assumption that the various versions exhibit independent failure modes.[8] The additional cost for producing and maintaining the extra versions is an important concern. For recovery blocks, the generation of the acceptance test is an art and is itself a potential source of faults.

We believe that the effectiveness and acceptability of software fault tolerance depends on the increased automation of the software development process. One benefit of this increased automation will be to offset the additional cost of providing the redundancy required for fault tolerance. Automation can also help manage the increased complexity inherent in a fault tolerant system. In addition, by introducing automation earlier in the software development cycle than is done currently, more information about the design process will be in machine-analyzable form. For example, executable specifications can be used to automate the error detection phase of a fault tolerant system as described in the next section. The information captured in the design phases may lead to fault tolerance techniques which go beyond the current techniques described in the literature. The reconfiguration of a computing system in order to meet critical mission objectives after a major system failure will require knowledge about the mission objectives and their interrelationships. Finally, automation may assist in the development of alternate designs which will exhibit different failure modes.

Although there has been extensive work in the areas of automatic programming, formal specification languages, and

Received Sept. 18, 1985; presented as Paper 85-6001 at the AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, Long Beach, CA, Oct. 21-23, 1985; revision received April 3, 1986.

*Assistant Professor, Department of Computer Science.

software fault tolerance, very little has been done to integrate these areas. This paper presents several potentially fruitful topics of investigation which may result in the automation of fault tolerant computing systems. Some initial results of research in the automatic generation of error detection for fault tolerance are also presented.

Automating Error Detection

The initial thrust of our research is in the automation of error detection from executable and formal specification languages. This area was chosen because of the pivotal role error detection plays in fault tolerant computing systems. Without the ability to detect errors in a timely fashion, the other phases of fault tolerance become irrelevant. Errors in software systems can be detected through the use of assertions about the state of the computation.† Assertions are conditions which should be true in the current state of the computation if the system is performing reliably. The presence of an error is indicated by an assertion which evaluates to false.

The use of assertions in both automatic programming and program verification is well established. Preconditions are assertions which are assumed to be true on entry into the computation. Postconditions are assertions which should be true on exit from the computation. Program verification attempts to prove that the program satisfies the postconditions upon termination, given that the preconditions are initially true.[9] In automatic programming, the construction of the proof generates the program as a by-product.[10]

The following three sections will discuss the coding of assertions, the generation of new assertions from specifications, and the generation of test cases using abstract data type specifications.

Coding of the Assertions

Assertions are specifications for detecting errors in a computing system. Just like any other specification, assertions must be translated into an executable code. Coding of assertions is a special case of the program synthesis problem, containing several significant features which differentiate it from the general case of program synthesis. To illustrate this with an example, let the specification of a sorting routine contain the assertion shown in Fig. 1. Using the deductive approach to automatic programming, a proof that the POST assertion follows from the PRE assertion is constructed. Since the POST condition is not, in general, true for LIST-IN, the proof requires an algorithm to generate LIST-OUT from LIST-IN. The algorithm is constructed in the course of the proof. In contrast, using this POST assertion for error detection will generate a different algorithm. In this example, checking that LIST-OUT is ordered is easier than sorting LIST-IN, but checking that it is a permutation of LIST-IN is quite involved.

A second distinctive feature of assertion code generation is that the choice of data representation for the data types in the assertion must be consistent with the data representation used in the program being monitored. In the preceding example, the same representation for "lists" must be used in constructing the code for the sorting program and the POST assertion.

```
PRE:    LIST_IN is a list of integers
POST:   LIST_OUT is a list of integers and
        LIST_OUT is ordered and
        LIST_OUT is a permutation of LIST_IN
```

Fig. 1 Specification for a sorting program.

†The term "computation" refers to any granularity of a computing unit including a single operation, statement, subroutine, package, process, or group of cooperating processes.

If the code for the assertions is to be synthesized automatically, then the decision about the data representation must be available in the machine. One means of capturing these decisions is in the form of a program transformation that transforms the abstract data type into its concrete representation.[11]

The third difference relates to the opportunities for weakening or strengthening the assertion in order to reduce the complexity of the synthesized code. In designing an error detection strategy, it may not be cost effective, or even feasible, to monitor the system for all possible errors. Thus a tradeoff exists between the amount of effort spent on error detection and the extent of the fault coverage. Modifying the assertion is one way to exercise this tradeoff. Most assertions represent both necessary and sufficient conditions for the correctness of some aspect of the program state. It may be possible to remove the necessary or sufficient property of the assertion. If the modified assertion is necessary but not sufficient, then certain states that would fail the original assertion may pass the modified one. However, no states that failed the modified assertion would have passed the original. On the other hand, if the modified assertion is sufficient but not necessary, then certain states that would fail the modified assertion could pass the original assertion. However, states that pass the modified assertion are guaranteed to pass the original one. This form of modified assertion can be used as a filter to identify potentially erroneous states that can be subjected to more careful testing, perhaps using the original assertion.

As an example of how an assertion could be modified, consider the condition in the POST assertion for the sorting routine that the LIST-OUT is a permutation of the LIST-IN (Fig. 1). The calculation of this assertion can be as complicated as the sort program itself. One possible weakening of this assertion is to require that the digital signature (checksum) of the input list and the output list be the same. Calculating the checksum of a message is frequently used for error detection during data transmission. The requirement for checksum equality is a necessary but not sufficient condition for permutation. However, the probability of a faulty program passing the weakened assertion can be made quite small with a good checksum algorithm. Not only is the checksum relatively easy to compute, it also allows the input list to be overwritten by the program once its checksum has been calculated. Thus the weakened assertion is more efficient in both time and space than the original assertion.

Although it seems more natural to weaken the original assertion so that the modified assertion is necessary but not sufficient, it is also possible to strengthen the original assertion. Again considering the permutation assertion, asserting that LIST-OUT is the same as LIST-IN strengthens the original assertion. This is a sufficient, but not necessary, condition for permutation. Since it is easy to compute, this modified assertion might be useful if almost every LIST-IN was already sorted. LIST-OUTs which fail the modified assertion could be scrutinized more closely, perhaps using the original assertion.

Generation of New Assertions from Specifications

As previously discussed, one source of assertions for error detection is from the PRE and POST conditions used to specify the program. Other assertions can be derived during the process of code synthesis, or by an analysis of the specification, or by observing the behavior of the system. A specification technique based on the concept of heterogeneous algebras[12] can be used to illustrate the generation of new assertions. This technique focuses on the algebraic structure of data types instead of assertions about the state of the computation. In the computing literature, specification techniques based on heterogeneous algebras are known as Abstract Data Types (ADT).[13] Figure 2 shows the specification of a set of integers using Abstract Data Types. The SYNTAX section specifies the types of the arguments and the value returned by each operation of the data type. The SEMANTICS is a set of axioms that

defines the meaning of the operations. An instance of all data type is determined by the sequence of operations that created it. The meaning of this sequence is determined by the equivalence class of data objects of which this particular instance is a member. For example, the operation "HAS" returns a Boolean data object. Therefore the sequence of operations "HAS(INSERT(NEWSET,1),1)" represents a Boolean data object. The meaning of this object is determined by applying axiom 4, which reduces the object to the equivalent Boolean object "True."

The algebraic specifications can be used to generate assertions about the data type operations. The generation of an assertion can be divided into two phases. In the first phase, a hypothesis about the behavior of the abstract data type is generated. This can be generated by a programmer, by a proof procedure used during the implementation of the abstract data type, or by a program that observes the behavior of the abstract data type and "learns" certain patters of behavior. In the second phase, it must be proven tha the hypothesis is a consequence of the axioms in the specification. Once the hypothesis has been proven, it can be used as an assertion about the state of the computation and inserted into the code as previously discussed. Even if the observed behavior is not invariant but is normative, it still may be useful to insert a check to trigger closer observation, audit tests, etc. when such abnormal, although possibly correct, behavior is observed.

As a simple example of assertion generation, consider the set specification given previously in Fig. 2. It can be observed that after some element "*i*" is deleted from a set, a test for the membership of "*i*" in that set will always be false [i.e., HAS(DELETE(*s,i*),*i*=False]. It is possible to generate this observation using some relatively simple heuristics based on the syntactic structure of the ADT. This observation is an assertion because it can be proved as a theorem that follows from the axioms of the specification shown in Fig. 2.[14]

Automatic Test Case Generation from Abstract Data Type Specifications

In designing test cases, both the input values and the corresponding output values must be generated. Generating the output values that correspond to the input values is not always easy. One of the advantages of ADT as a specification technique is its ability to be executed directly using the axioms as rewrite rules.[15] This process allows the specification to be used as a testing oracle.[16] For example, any implementation of sets must return "True" for the sequence of operations HAS (INSERT(*s,i*),*i*), where "*s*" is an arbitrary set and "*i*" is an arbitrary integer. This behavior is a consequence of applying axiom 4 of Fig. 2 as a rewrite rule to reduce the sequence of operations to a Boolean value. The generation of test cases can be automated by an analysis of the ADT specification.[17]

There is a strong relationship between assertion generation and test case generation. A particular test case is an assertion about the relationship between specific input and output values. Symbolic testing generalizes this relationship and thus is strongly related to assertion generation. Conversely, the checking of assertions is a runtime test. Generally, assertions cannot be sensitive to the past history of execution. For example, the value of HAS(*s,i*) at an arbitrary point in the execution depends on the past history of insertions and deletions. During preproduction testing, this history is determined by the testing system itself. Note, however, that the value of HAS(*s,i*) is known in the special case in which the set "*s*" is the one from which "*i*" has just been deleted or into which "*i*" has just been inserted. In this special case, HAS(*s,i*) can be used as an acceptance test (assertion) for the DELETE and INSERT operations, respectively.

TYPE *iset*

SYNTAX

NEWSET	→ iset
INSERT(iset,integer)	→ iset
DELETE(iset,integer)	→ iset
HAS(iset,integer)	→ boolean
ISEMPTY(iset)	→ boolean

SEMANTICS

declare s:iset;i,j:integer;

```
1)  DELETE(NEWSET,i)        ⇒  NEWSET
2)  DELETE(INSERT(s,i),j)   ⇒
        if i = j then
            DELETE(s,j)
        else
            INSERT(DELETE(s,j),i)
3)  HAS(NEWSET,i)           ⇒  False
4)  HAS(INSERT(s,i),j)      ⇒
        if i=j then True
        else HAS(s,i)
5)  ISNEW(NEWSET)           ⇒  True
6)  ISNEW(INSERT(s,i))      ⇒  False
```

Fig. 2 Specification of a set of intergers.

Continued Service

While error detection is a crucial element of any fault tolerant system, it is necessary to be able to recover from the error and continue service if the system is to survive the fault. The most common techniques for software fault tolerance employ the use of redundant versions of the program to mask the effect of the fault. A critical assumption with respect to software design faults is that the various versions of the program will exhibit independent failure modes. The use of independently generated version has been called "design diversity" by Avizienis.[4] However, the design diversity is a hoped for side effect of isolating the various groups producing the different versions. Automatic programming offers the possibility of true design diversity through the deliberate development of alternate versions of a program. The programming process can be viewed as a search through a tree of all possible solutions.[18] Each node in the tree is a decision point which binds some portion of the abstract specification to a more concrete implementation. A sequence of decisions defines a path through the tree from the abstract specification to its implementation. Because of the potential diversity of decision paths, there may be many programs which will implement the same specification. Some of these programs may be equivalent functionally but exhibit different performance characteristics. It is also possible that some of the programs are functionally different. Each decision is made under a set of assumptions about the appropriateness of the program as a model of the underlying problem. The set of assumptions along the entire design path defines a support set for the final program. Two versions with different support sets could behave differently under the same conditions.

Since an automatic programming system makes each decision point explicit, it is possible to explore alternate decision paths. These alternate paths may lead to truly diverse designs which model the specification under different sets of assumptions. As a simple example, consider the decision to use single vs double precision numbers. In the absence of any requirement in the specification, it may be reasonable to generate two versions, one using single precision arithmetic and the other using double precision. The single precision version may be able to handle 99.9% of all cases, with the alternate double precision version used to handle the relatively rare combination of circumstances that require more precision. It should be noted that even if there is a requirement in the specification regarding the precision, that requirement may be based on

some assumptions about expected ranges of the input data values. Those assumptions may not be justified in certain feasible but rarely occuring combinations of data values.

Concluding Remarks

The production and operation of a class of software systems, referred to in this paper as autonomous survivable computing systems, will require new approaches to software development. These approaches will require the integration of several existing areas of research and development in computer science. These areas include automatic programming, software fault tolerance, formal and executable specification languages, program verification, and knowledge-based software engineering. In this paper, two aspects of software fault tolerance, the detection of erroneous states of the computation and continued service after the detection of errors, were discussed.

Preliminary research in the automatic detection of errors indicates the importance of a formal, executable specification. Using algebraic specification of abstract data types, it is possible to generate specific test cases as well as assertions about the general behavior of the proposed system. Because an algebraic specification has formal deductive semantics, it is possible to prove that the assertions follow as theorems from the specification. These assertions could be inserted into the final system to monitor its correct execution. Weakening or strengthening the assertions to make them easier to compute appears to be an original suggestion.

A second aspect of fault tolerant software systems discussed in this paper is the ability to recover from any errors which are manifested during operation. In order to recover from errors, alternate models of the computation with different failure modes must be developed. Design diversity, needed for continued service, may be possible by exploring alternate design paths in the program synthesis tree. This suggestion, that the development of diverse designs be a deliberate and conscious pursuit, could be integrated into automatic program synthesis approaches that search for implementations to satisfy a given specification. The deliberate pursuit of diverse designs may also prove fruitful in the manual generation of multiversion software. However, much more research needs to be done on the relationship between individual design decisions and the failure modes of software systems.

Increased automation of the software development process will be required to make the development of autonomous survivable systems feasible. It is important to capture in machine-analyzable form knowledge at all levels about the system requirements and the rationale behind the design decisions. The task of designing, implementing, and operating highly reliable computing systems is immense and will require the synthesis of techniques from many areas of investigation. The issues of fault tolerance must be addressed at all levels of system design and implementation if we are to build truly reliable computer systems.

Acknowledgments

I would like to thank Dave Eckhardt at NASA Langley Research Center, Hampton, VA for his continued support of this work. The motivation for this research came from his desire to study the relationships between automatic programming and software fault tolerance. His many ideas and suggestions have served as inspiration throughout this work. This work was sponsored in part by the NASA Langley Research Center under Grant NAG-1-439.

References

[1]Wild, C. and Toida, S. "Potential Uses of Automatic Programming Technology in the Production of Ultra-Reliable Computing Systems," Dept. of Computer Science, Old Dominion University, Norfolk, VA, TR-85-007, March 1985.

[2]Anderson, T. and Lee, P.A., *Fault Tolerance: Principles and Practice,* Prentice-Hall International, Englewood Cliffs, NJ, 1981.

[3]Raney, L., "The Use of Fault-Tolerant Software for Flight Control Systems," *NAECON,* Institute of Electrical and Electronics Engineers, May 1983, pp. 1287-1291.

[4]Avizienis, A. and Kelly, J., "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer,* Vol. 17, August 1984, pp. 67-80.

[5]Slivinski, T., Goldberg, J., Anderson, T., Kelly, J., Hitt, E., Webb, J., Wild C., and Levitt, K., "Study of Fault Tolerant Software," Mandex, Inc., Springfield, VA, Final Report for NASA Contract NAS1-17412, May 1984.

[6]Chen, L. and Avizienis, A., "N-Version Programming: A Fault Tolerant Approach to Reliability of Software Operation," *Digest of Papers, Fault Tolerant Computing Symposium* (FTCS-8), June 1978, pp. 3-9.

[7]Randell, B., "System Structure for Software Fault Tolerance," *Programming Methodology,* edited by D. Gries, Springer-Verlag, New York, 1978, pp. 362-387.

[8]Knight, J. and Leveson, N., "Independence in Multiversion Programming," *IEEE Transactions on Software Engineering,* Vol. SE-12, Jan. 1986, pp. 96-109.

[9]Loeckx, J. and Sieber, K., *The Foundations of Program Verification,* Wiley, New York, 1984.

[10]Manna, Z. and Waldinger, R., "A Deductive Approach to Program Synthesis," *Automatic Program Construction Techniques,* edited by A. Biermann, G. Guiho, and Y. Kodratoff, Macmillan, New York, 1984, pp. 33-68.

[11]Burstall, R.M. and Darlington, J., "A Transformational System for Developing Recursive Programs," *Journal of the ACM,* Vol. 24, Jan. 1977, pp. 44-67.

[12]Birkhoff, G. and Lipson, J.D., "Heterogenous Algebras," *Journal of Combinatorial Theory,* Vol. 8, 1970, pp. 115-133.

[13]Berztiss, A. and Thatte, S., "Specification and Implementation of Abstract Data Types," *Advances in Computers,* edited by M.C. Yovits, Academic Press, 1983, pp. 295-353.

[14]Wild, C. and Pang, A. "Generation of Rules of Behavior from an Analysis of Abstract Data Type Specifications," Department of Computer Science, Old Dominion University, Norfolk, VA, in preparation.

[15]Guttag, J., Horowitz, E., and Musser, D., "Abstract Data Types and Software Validation," *Communication of the ACM,* Vol. 21, Dec. 1978, pp. 1048-1064.

[16]Gannon, J., McMullin, P., and Hamlet, R., "Data Abstraction - Implementation, Specification and Testing," *ACME Transactions on Programming, Languages and Systems,* Vol. 3, 1981, pp. 211-223.

[17]Wild, C., Eckhardt, D., Pang, A., and Sundararajan, S., "Analysis of Executable Specifications for Testing and Monitoring Abstract Data Types," Department of Computer Science, Old Dominion University, Norfolk, VA, TR-86-006, March 1986.

[18]Smith, D., Kotik, G., and Westfold, S., "Research on Knowledge-Based Software Environments at Kestrel Institute," *IEEE Transactions on Software Engineering,* Vol. SE-11, Nov. 1985, pp. 1278-1295.