

# A Formal Method with TUG for Developing Reliable Programs

Chia-Chu Chiang

Department of Computer Science  
University of Arkansas at Little Rock  
2801 South University Avenue  
Little Rock, Arkansas 72204-1099  
E-mail: cxchiang@ualr.edu

**Abstract** – A formal method with a formal specification language, called TUG (Tree Unified with Grammar), is presented to model software development process. The use of the method in conjunction with guidance and traceability makes the method singularly different from existing formal methods and also facilitates developers in building more reliable programs.

## I. INTRODUCTION

Formal methods provide a rigorous means of developing software systems. However, many formal methods support software development only in certain phases, but few support the full cycle of software development [1]. In this paper, a formal method with a formal specification language, called TUG, is presented to support a system to be developed in an integration of software development activities such as conventional software development, executable specification, rapid prototyping, specification reuse, and analysis of the system. The method improves the software development process by detecting errors early. The pursuit of these improvements contributes to the production of reliable programs.

## II. SOFTWARE DEVELOPMENT WITH TUG

The software development process with TUG is shown in Figure 1.

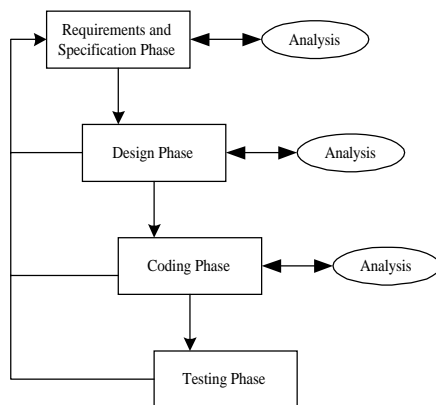


Figure 1. Software development with TUG

A software system is mainly developed through a sequence of phases such as requirements specification, design, implementation, and testing. The method provides analysis techniques for detecting errors in every phase. Two main analysis techniques are used for detecting errors: static analysis and dynamic analysis. Static analysis detects the errors without executing the software. Dynamic analysis detects the errors by executing the software.

### A. Requirements and Specification Phase

The method with TUG supports for operational specification and rapid prototyping that allow software developers to detect requirements and specification errors by exercising a TUG specification or a prototype generated from the specification. The method also provides a static analysis technique to verify the specifications against the user requirements using theorem proving. A TUG specification is transformed into a set of Horn clauses. With the set of clauses, the specification is ready for proof. First, the user makes a claim about the specification. With the negation of a user claim, the set of clauses is used to show whether the negation of the user claim is a logical consequence of the set of clauses through theorem proving.

Software reuse should be engineered as early as possible in the software development process. TUG provides language constructs for polymorphism, overloading, parameterized templates, root type, subtypes, renaming, extending, and sharing to foster specification reuse. Two approaches – vertical reuse and horizontal reuse are provided to build scalable parameterized templates hierarchically without replication [2]. Vertical reuse makes a lower level template inherit all features from its upper level template. Horizontal reuse makes a template reusable within the same level of abstraction. The method facilitates software developers to build reusable specifications following domain analysis and analogical reasoning.

### B. Design Phase

TUG was developed to construct a specification through three structuring constructs: sequence, selection, and iteration [3]. Therefore, a TUG specification can be mapped

into a set of structured diagrams in which each structure diagram corresponds to a specification module. A proof technique can be used for analyzing a design to assure that the design is correct against the specification. First, a design is constructed using the structured design method. The design diagram is composed of a set of design blocks. A block may denote a structure notation (sequence, if, and while), a single statement, a set of statements, or a complete program. A block also specifies its pre- and post-conditions. The blocks with proof rules are used to show the partial correctness of the design using Hoare's axioms and rules [4]. The method applies the loop approach to the structured design for proving termination [5].

### C. Coding Phase

A structured design is mapped into a set of structured programs in which each program module corresponds to a structured diagram. Based on the specification, each module should have explicit functions to be performed and can be supplied with a pair of pre- and post-conditions in terms of the functions. Analysis starts on a pair of pre- and post-conditions with respect to a module. Top-down proofs can be carried on from the system to the subsystems and modules. Hoare's axioms [4] and the loop approach [5] are used to show the total correctness of programs.

### D. Testing Phase

The method with TUG provides a specification-based testing technique for generating test cases from TUG specifications. However, there may be an infinite set of test cases from which a choice must be made. In order to make a finite set of test cases; a test strategy may be developed to facilitate the test case selection. The technique for automated test generation from TUG specifications is still under development.

## III. NEW ASPECT OF THE METHOD FOR MAKING SOFTWARE MORE RELIABLE

Formal methods such as Larch [6], [7], VDM [8], and Z [9], [10] provide a rigorous method for developing software systems. Unfortunately, developers are reluctant to use formal methods because of the mathematical notation or formality. We believe that formal methods should not only provide a more rigorous means of detecting errors but should also provide guidance in software development. The method we presented in this paper not only provides a means of detecting errors, but also provides guidance in constructing and proving software in terms of the specification's structuring notation. Developers should feel more comfortable developing design and software, if provided with guidance.

The method with TUG also supports the traceability in the maintenance phase. Software development activities

such as specification, design, coding, and proof are based on structuring constructs and through the use of structuring constructs, a linkage is made between specification, design, code, and proof. Whenever there is a change in the user's requirements, the impact and changes can be located and traced in the documents in terms of the structuring constructs. The aids of guidance and traceability improve the method with TUG for making software more reliable. In addition, providing guidance and traceability for the developer in constructing software makes this method singularly different from existing formal methods such as Larch, VDM, and Z.

## IV. SUMMARY

The quality of software depends mainly on the effectiveness of the software development activities. In this paper, a formal method with a formal specification language, called TUG, was presented to support a system to be developed in an integration of software development activities such as conventional software development, executable specification, rapid prototyping, specification reuse, and analysis of the system. The method provides testing and proof techniques for software engineers to detect errors in every phase of the software development process. Whenever there is any doubt concerning testing, proofs should resolve the doubt and vice versa.

## REFERENCES

- [1] S.-Y. Liu, J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba, "SOFL: A Formal Engineering Methodology for Industrial Applications," *IEEE Transactions on Software Engineering*, Vol. 24, No. 1, January 1998, pp. 24-45.
- [2] C.-C. Chiang and D. Neubart, "Constructing Reusable Specifications through Analogy," *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC'99)*, February 28-March 2, 1999, San Antonio: Texas, USA, pp. 586-592.
- [3] E. W. Dijkstra, *A Discipline of Programming* Prentice-Hall, Englewood, New Jersey, 1976.
- [4] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, Vol. 12, No. 10, October 1969, pp. 576-580, 583.
- [5] S. Katz and Z. Manna, "A closer look at termination," *Acta Informatica*. Vol. 4, No. 4, 1975, pp. 333-352.
- [6] J. V. Guttag, J. J. Horning, and J. M. Wing, "The Larch Family of Specification Languages," *IEEE Software*, Vol. 2, No. 5, September 1985, pp. 24-36.
- [7] J. V. Guttag and J. J. Horning, "A Larch Shared Language Handbook," *Science of Computer Programming*, Vol. 6, No. 2, March 1986, pp. 135-157.
- [8] C. B. Jones, *Systematic Software Development using VDM*, Prentice-Hall International, London, U.K., 1986.
- [9] J. M. Spivey, *Understanding Z*, Cambridge University Press, Cambridge: United Kingdom, 1988.
- [10] J. M. Spivey, *The Z Notation – A Reference Manual*, 2<sup>nd</sup> Ed., Prentice-Hall International, London: United Kingdom, 1992.