ATP 46

Internal Note 4/30/77
Certifiable Minicomputer Project
The University of Texas at Austin

THEOREM PROVING IN A PROGRAM VERIFICATION CONTEXT

Mark S. Moriconi, Mabry Tyson, Richard M. Cohen
The University of Texas at Austin

Introduction

This paper considers the symbiosis between theorem proving and program verification and focuses on the way this interrelationship has affected the design of a program verification system currently under development.

Verifications are typically carried out in less well-understood domains (e.g., communications processing) than highly-developed areas of traditional mathematics. For example, it is commonplace in verification -- as in the analog of developing new mathematical theories -- to introduce lemmas or definitions, revise them for any of several reasons, then try proofs again. Because of this, several important problems arise.

Verifying a program may involve proving hundreds of theorems and assuming hundreds of properties for use in proofs. These properties are intended to formally characterize the verification domain and are frequently revised throughout a verification. Revisions are necessary (1) when a proof attempt fails and (2) when proofs succeed due to an incorrect characterization of the problem domain. The *proof management problem* is to decide what proofs are affected by revisions, and thereby avoid redoing any still-valid proofs. In order to do this, the verification system has a proof manager that keeps track of proof dependencies.

Closely scrutinizing proofs aids in discovering what revisions are necessary. As a result, our theorem prover addresses the *proof presentation problem* of collecting and displaying detailed records of proofs that make explicit

----

Part of the first author's work was done at USC Information Sciences Institute.

Authors' address: Certifiable Minicomputer Project, University of Texas at Austin, Austin, Texas 78712.

steps in derivations along with their justifications.

After the user makes revisions, the proof manager supplies the theorem prover with the proofs that need to be (partially) redone and the revised properties. The theorem proving _re-proof problem_ is how to retain still-valid parts of affected proofs.

We discuss these and other issues in the context of this verification system and its theorem prover. Examples will be given.

System Overview

The purpose of a program verification system is to automate the task of proving the consistency between a computer program and specifications or assertions describing what the program is supposed to do. Program text is supplied together with assertions describing the program. Verification involves demonstrating that programs meet their specifications. This is done by proving theorems, called verification conditions (VCs), derived from the program text.

This system evolved from the verifier described in Good, London, and Bledsoe [75], and is discussed further in Moriconi [77]. The theorem prover is an extension of an already highly-developed program discussed in Bledsoe and Tyson [75a]. Programs written in the language Gypsy [Ambler, Good, and Burger 76] are verified. Language features include concurrency, data abstraction, and run-time error recovery.

Impact of Theorem Proving on Verification System Design

Several principles employed in the verification system design are motivated in part by theorem proving considerations. Some are outlined below.

Information management. Verifications involve developing an interrelated collection of information that includes programs, specifications, VCs, assumed properties, and proofs. Storing and relating this information in a single data base enables the theorem prover, for example, to access any needed information without knowledge of its origin or of its representation details. For example, the prover's typelist mechanism handles integer-valued variables [Bledsoe and Tyson 75b]. The prover, therefore, queries the data base for type information before entering a variable into the typelist. No knowledge is required of the prover concerning context or symbol table representation.

Proof management. Because theorem provers deal with individual theorems, a proof manager is needed to maintain the integrity of a collection of proofs that arises when verifying a large program. Doing this in a verification context raises several questions discussed in Moriconi [77], such as determining what is affected by revisions, and determining when and how inconsistencies can be resolved. We focus here on the mechanism for keeping track of what properties are used in each proof. The proof manager can then identify which proofs need to be redone if any property is changed and which proofs remain valid.

The user can revise properties whenever convenient. This flexibility is useful, for example, when the user discovers at some point in a proof that a new property is required. When he supplies the needed property to the prover, the proof manager stores it in the data base for subsequent reference and updates the data base appropriately.

Size of VCs. VCs are often quite large and bulky, hindering effective user analysis. For example, some VCs that arise from concurrency in communications processing examples are 50 to 100 lines long. Limiting their size significantly aids the interactive proving process. We address this problem by simplifying VCs as they are generated and by deferring the expansion of specifications from called programs in both specifications and executable code until the actual proof has begun. These system features tend to reduce the size of VCs, enhance readability, and keep them in terms of user-defined abstractions.

Deferred expansion is an important system feature. The VC generator inserts only references to specifications of called programs, instead of the specifications themselves. Then, complete specifications, or parts of specifications, are expanded automatically or interactively as needed during the proof. The data base, proof manager, and prover interact to perform expansions.

Impact of Verification on Theorem Prover Design

If an attempt to prove a VC fails, the user revises the program, its specifications, or assumed properties before trying the proof again. Understanding why proofs fail suggests what to revise. See von Henke and Luckham [75] and Katz and Manna [75] for related work. On the other hand, successfully proving a VC does not necessarily imply that the description of the domain is correct. For example, individual properties that appear to accurately characterize the domain may subtly interact to cause unintended inferences; or a proved subgoal may be intuitively incorrect in the domain. Closely scrutinizing detailed records of proofs aids in identifying the cause of such anomalies.

Analysis of proofs is aided by our use of an interactive, natural-deduction theorem prover. An interactive policy is useful when attempting to prove a currently unproveable VC because the user can examine the context of the failure. Often all that is needed is to supply additional properties. A natural-deduction proof strategy is important because dialogs and records of proofs are in a format that is natural and convenient to the user.

Proof presentation. Since successful proofs may depend upon properties that incorrectly describe the domain, easily-understandable records of proofs -- displaying how properties are used -- are important as documentation for verifications to be credible. Thus, the prover collects a detailed record of proofs that makes explicit both built-in and user-supplied assumptions. This record is then stored by the proof manager for subsequent reference. The user views these proofs with a dual-mode proof display facility. He has the option of interactively directing presentations by requesting the desired amount of detail at selected steps, or of having a completely automatic presentation by specifying initially the amount of detail desired throughout.

Re-proofs. Recall that when revisions are made the proof manager preserves still-valid proofs, then invokes the prover with the affected proofs and the revised properties. The prover is now confronted with the problem of retaining still-valid parts of these affected proofs. Although this problem may appear unrelated to the proof presentation problem, they are in fact highly interrelated. Proofs of affected subgoals must be consistent with other parts of the original proof. We, therefore, use these representations of proofs to supply the necessary context. For example, the prover must be aware of all substitutions made in proofs and also the context in which they were made because substitution conflicts may invalidate previous deductions.

The prover currently accepts re-proofs when the substitutions made in a subgoal's new proof subsume those made in its previous proof. We are investigating other parts of this problem within the logic of our prover.

Acknowledgements

We are indebted to W.W. Bledsoe and Donald I. Good for their support and numerous contributions to this work.

The research described here was supported in part by the National Science Foundation (Grant DCR 74-12866) and the Defense Advanced Research Projects Agency (Contract

DAHC-15-72-C-0308 and Order 3286). The views expressed are those of the authors.

References

A.L. Ambler, D.I. Good, W.F. Burger [76], Report on the Language Gypsy, ICSCA-CMP-1, The University of Texas at Austin, 1976.

W.W. Bledsoe and M. Tyson [75a], The UT Interactive Prover, University of Texas at Austin Mathematics Department Memo ATP-17, May 1975.

W.W. Bledsoe and M. Tyson [75b], Typing and Proof by Cases in Program Verification, University of Texas at Austin Mathematics Department Memo ATP-15, May 1975.

F.W. von Henke and D.C. Luckham [75], A Methodology for Verifying Programs, Proc. of International Conference on Reliable Software, April 1975, 156-164.

S. Katz and Z. Manna [75], Towards Automatic Debugging of Programs, Proc. of International Conference on Reliable Software, April 1975, 143-155.

M.S. Moriconi [77], An Interactive System for Incremental Program Design and Verification, Ph.D. thesis, University of Texas at Austin (in preparation).