N96- 10030

INTRODUCTION TO PENELOPE

59514
p. 9

David Guaspari

Odyssey Research Associates

A formal program verification is a (mathematical) proof that a program executed according to its intended model meets some specification. This proves that the algorithm defined by the program is *correct* in the precise technical sense of being consistent with a particular specification. A program correct in this sense is free from a large and important class of errors, even though its behavior may still produce unintended results---either because the implementation of the programming language itself does not match the model of execution, or because the specification does not correctly express the user's intentions.

Penelope is a prototype system for interactively developing and verifying programs that are written in a rich subset of sequential Ada. Penelope can be used to develop a program and its correctness proof incrementally, and in concert with one another. Incrementality is used in a number of ways to help make verification more tractable and more productive. For example, if an already-verified program is modified, one can attempt to prove the modified version by replaying and modifying the original verification.

Penelope's specification language, Larch/Ada, belongs to the family of Larch interface languages. Larch/Ada scales up properly, in the sense that it is demonstrably sound to decompose a system hierarchically and reason locally about the implementation of each piece.

Penelope has been applied in various demonstration projects---for specification (guidance control, distributed operating system), verification (of off-the-shelf code), and formal development (by non-expert as well as expert users). Some features of Penelope have been embodied in AdaWise, a lint-like non-interactive tool that warns of the potential for certain dynamic semantic errors in Ada programs.

Introduction to Penelope

David Guaspari
NASA Formal Methods Workshop
May 10 - 12, 1995

Odyssey Research Associates
301 Dates Drive
Ithaca, NY 14850-1326
(607) 277-2020
davidg@oracorp.com

Goals of the Penelope project

- Define the semantics of a large Ada subset
- Define an Ada specification language (Larch/Ada)
- Implement automated support for reasoning about specifications and implementation (Penelope)
- Apply Penelope

Results

- Mathematical goals met
- Penelope supports a non-trivial subset of our model
- Applications have been demonstrations
- Spin-offs
  - "Lightweight" Ada tools (AdaWise)
  - Carryover to Ada95 work (LPT)
  - Penelope as a teaching tool

Outline of this talk

- Background (modeling Ada)
- The Larch/Ada specification language
- The Penelope system
- Some Penelope and AdaWise applications

Semantics of Ada

- Complicated parts
  - static semantics
  - concurrency
- Dynamic semantics of sequential Ada is clean
  - strong typing
  - run-time constraint checking
  - disciplined use pointers
  - disciplined use of exceptions
  - information hiding
  - standardizes semantics often left implementation-dependent
- Dark corners of sequential Ada:
  - arbitrary choices left underdetermined
  - interactions of optimization and exceptions

Definitions of Ada

- In the Ada culture
  - Officially: LRM + AI's
  - Ad hoc standard: ACVC
- Formal definitions
  - Formal definition of sequential Ada80
  - AdaEd interpreter (in SetL)
  - DDC + CRAI definition of Ada83 (virtually complete)
- Formal definitions of subsets
  - ORA
  - CLInc
  - Aerospace Corporation
  - Program Validation, Ltd.
  - ProSpectra

Modeling our Ada subset

- Eliminate almost all dark corners by static semantic checks
  - for "improper" aliasing
  - for "undisciplined" side effects
- Disallow optimizations sanctioned by RM 11.6.
- Ignore resource limitations (storage error, numeric overflow).

Definitional technique: denotational semantics, predicate transformers.

Subset covered by the model

- Nearly all non-pathological uses of the following:
  - All sequential types (including floating point)
  - All sequential statements
  - Exception-raising and -handling
  - Subprograms (including side-effects, recursion, global variables)
  - Packages
  - Generics

What a Penelope proof proves

- Hypotheses on compiler
  - No section 11.6 optimizations
  - No optional program_error
- Hypotheses on executions
  - No storage error
  - No numeric overflow
- Currently unchecked
  - Consistency conditions on theories

For allowed executions on allowed compilers: if initial conditions satisfied then termination implies exit conditions satisfied.

Predicate transformer semantics: a "logical" form of symbolic execution

```
--|      ? --(1) precondition (sought)
x := x+1;
--| y < x --(2) postcondition (given)
```

Question: What must be true at (1) to guarantee "`y < x`" will be true at (2)?

Answer: "`y < x+1`"

Answer obtainable by symbolic manipulation: substitute "`x+1`" for "`x`" in "`y < x`"

Analogy between predicate transformers

`wlp(S,__): predicate->predicate`

which take postcondition to precondition, and continuation semantics

`𝓜[[S]]: continuation -> continuation`

which takes (post)continuation to (pre)continuation.

This connection is developed in work by Wolfgang Polak.

Systematically associate computational entities with symbolic entities, e.g.:

Computational	Symbolic
value	term
answer	{true, false}
continuation	predicate

Derived predicate transformer semantics can be proven sound for the corresponding denotational model.

Penelope supports a special "asymptotic" floating point semantics

- Models computation in the limit as machine accuracy improves
- Purely algebraic reasoning about approximate operations – approximate equality, etc. (no epsilons and deltas)
- Highlights logical errors in specifying and coding with discontinuous operations (such as floating point comparisons)

Larch/Ada

- Specifies sequential Ada
  - Assertional (entry-exit, invariants, ...)
  - Partial correctness
- Unit of specification is the compilation unit
- Annotations of bodies hidden from clients
- "Two-tiered" semantics

Two-tiered specifications:

- Mathematical component
  - An environment of mathematical definitions
- Interface component
  - Describes behavior in terms of environment

Informal example of a two-tiered specification

```
function plus(x,y: integer) return
   integer;
```

- Mathematical component
  - Defines sort Int, the infinite collection of mathematical integers
  - Defines the basic mathematical operations on Int (+, *, <, ...)
- Interface component
  - Type integer is *based on* sort Int
  - Behavior of plus
    - Entry: No assumptions about state of parameters
    - Normal exit: Return x+y (mathematical sum), if termination is normal
    - Exceptional exit: raise error iff, on entry $x+y < -(2^{32})$ or $x+y > 2^{32} - 1$ (all operations mathematical)
    - No side effects

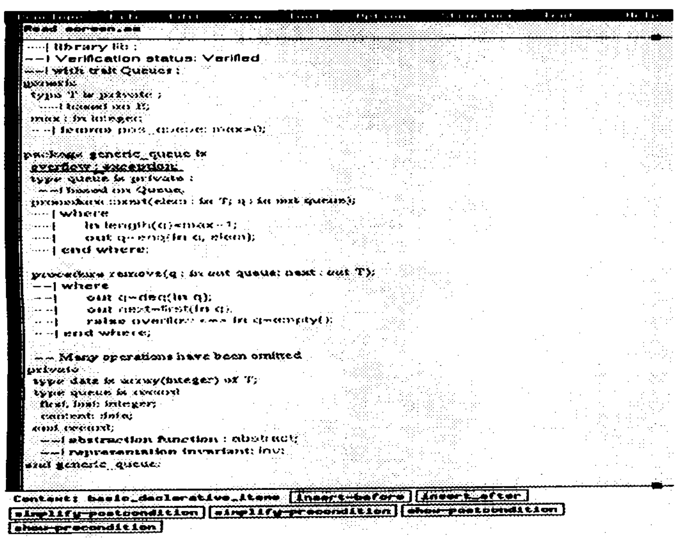
Mathematics defined in Larch Shared Language (LSL)

- Designed by John Guttag (MIT), Jim Horning (DEC SRC)
- Notation for ordinary mathematics
  - Sorts (sets of values)
  - Operations on sorts
  - Logical operations
  - Strongly sorted (i.e., typed)
- "Trait" – A modular way to describe theories
  - axioms
  - assumptions
  - conclusions
- Independent of program language

We have extended Larch

The Penelope System

- Editor for Ada programs with Larch/Ada annotations
- Automated support for developing (extended) Larch Shared Language
- Automated support for assertional reasoning about code
  - Incrementally generates verification conditions (VCs)
  - VCs are statements in ordinary logic
  - Supports Dijkstra's goal-directed style of program development
- Permits incremental reasoning
  - Factors proofs
  - Proofs are replayable
- Simple library, pretty-printing
- Implemented with the Synthesizer Generator

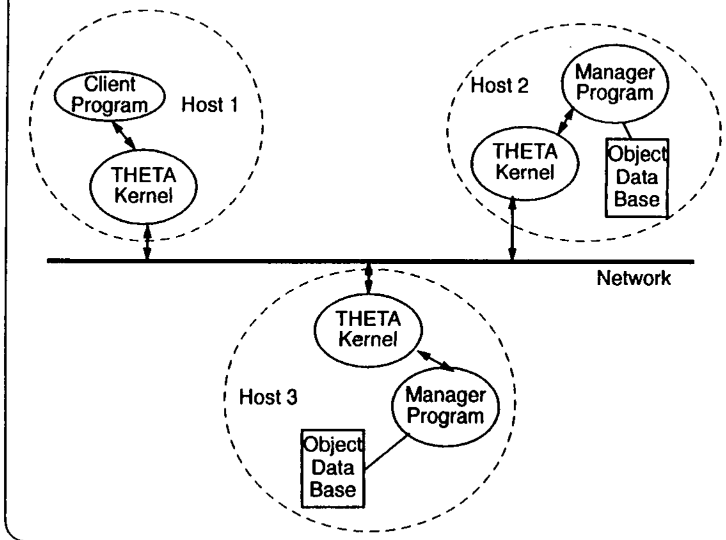


Some applications

- Theta kernel (security, specification exercise)
- COTS code (calendar package)
- Use by non-expert user (generic sets package for 777)
- Applications of AdaWise (to itself, Theta, repository code)
- Orbit calculations (floating point)
- Specification exercises (guidance control, flight management)

The Theta kernel

- Authentication and message routing for a trusted distributed heterogeneous operating system
- Penelope used to record high-level design, perform simple refinement proofs
- Described in "Applications of Formal Methods", edited by M.G. Hinchey and J.P. Bowen, Prentice Hall International Series in Computer Science, Hemel Hempstead, 1995.



COTS code (calendar package)

- Undocumented assumptions
  - No impossible dates are entered
  - Strings are numbered from 1
- Years represented by last two digits
- Careless arithmetic -
  - E.g., `Natural((day mod 7) - 1)`vs. `Natural((day - 1) mod 7)`

Non-expert user (generic sets package)

- User expert in testing
- Training:
  - Four days of supervised practice (at ORA)
  - Regular e-mail exchanges
- Difficulties
  - The "usual" FM problems
  - Penelope is not robust
  - Currently supported subset required workarounds
- User succeed in specifying and proving generic sets package (roughly 6 weeks of work from scratch)

AdaWise -- "Lightweight tools"

- Push-button (lint-like)
- Warns of potential for certain run-time problems
- Conservative (no warning implies no problem)
- Built on ASIS (runs with RISCAda, SunAda, Rational Apex)

Funded by Air Force Rome Laboratories and ARPA (STARS)

Properties checked

- Improper aliasing
- Incorrect order dependence
- Access to undefined scalars (under development)

Experience (Theta, repository code)

- Elaboration order dependences are common (mostly a portability problem)
- Obscure aliasing problems fairly common
- False warnings often point to doubtful coding practices
- Size of individual examples: up to 50-60 modules, 18,000 SLOC

Alias Warning: Non-Scalar Parameters

- Forms Generator:

```
**** form_executor lines  128 to  130:
            FORM_MANAGER.GET_FIELD_INFO
                (FIELD, NAME, POSITION, LENGTH, RENDITION,
                CHAR_LIMITS, VALUE, VALUE, MODE)

>> Parameters:  7 and  8 are potential ALIASES
>> (potential ORDER of COPY OUT error) and
>> (potential ERRONEOUS EXECUTION)
```

Alias Warning: Non-Scalar Parameters (cont'd)

```
procedure GET_FIELD_INFO( ... ;
                          INIT_VALUE : out FIELD_VALUE;
                          VALUE : out FIELD_VALUE;
                          ...) is
begin
   ...
   VALUE := FIELD.VALUE;
   MODE := FIELD.MODE;
exception
   ...
end GET_FIELD_INFO;
```


Further Work

- Tractability of constraint checking, definedness checking
- Packages with state
- Specification of termination proofs
- Improved modularity
- Full support for generics
- Concurrency

omit

Session 6: Hardware Systems

Paul Miner, Chair

- **The Formal Verification Technology Used on AAMP5**, by *Mandayam Srivas*, SRI International
- **Specification and Verification of VHDL Designs**, by *Damir Jamsek*, Odyssey Research Associates
- **Derivational Reasoning System**, by *Bhaskar Bose*, Derivation Systems Inc.