N96- 10029

375/3

p. 10

FORMAL METHODS IN THE DESIGN OF ADA 95

David Guaspari

Odyssey Research Associates

Formal, mathematical methods are most useful when applied early in the design and implementation of a software system---that, at least, is the familiar refrain. I will report on a modest effort to apply formal methods at the earliest possible stage, namely, in the design of the Ada 95 programming language itself. This talk is an ``experience report" that provides brief case studies illustrating the kinds of problems we worked on, how we approached them, and the extent (if any) to which the results proved useful. It also derives some lessons and suggestions for those undertaking future projects of this kind

Ada 95 is the first revision of the standard for the Ada programming language. The revision began in 1988, when the Ada Joint Programming Office first asked the Ada Board to recommend a plan for revising the Ada standard. The first step in the revision was to solicit criticisms of Ada 83. A set of requirements for the new language standard, based on those criticisms, was published in 1990. A small design team, the Mapping Revision Team (MRT), became exclusively responsible for revising the language standard to satisfy those requirements. The MRT, from Intermetrics, is led by S. Tucker Taft.

The work of the MRT was regularly subject to independent review and criticism by a committee of Distinguished Reviewers and by several advisory teams---for example, by two User/Implementor teams, each consisting of an industrial user (attempting to make significant use of the new language on a realistic application) and a compiler vendor (undertaking, experimentally, to modify its current implementation in order to provide the necessary new features). One novel decision established the Language Precision Team (LPT), which investigated language proposals from a mathematical point of view.

The LPT applied formal mathematical analysis to help improve the design of Ada 95 (e.g., by clarifying the language proposals) and to help promote its acceptance (e.g., by identifying a verifiable subset that would meet the needs of safety-critical applications). The first LPT project, which ran from the fall of 1990 until the end of 1992, produced studies of several language issues: optimization, sharing and storage, tasking and protected records, overload resolution, the floating point model, distribution, program errors, and object-oriented programming. The second LPT project, in 1994, formally modeled the dynamic semantics of a large part of the (almost) final language definition, looking especially for interactions between language features.

Formal methods in the design of Ada95

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

Revising Ada

- Requirements solicited, published in 1990
- Mapping Revision Team (MRT)
  - Small design team (as with Ada83)
  - Led by S. Tucker Taft (Intermetrics)
- Advisory groups
  - Distinguished Reviewers
  - User/Implementor Teams
  - Language Precision Team (1991-1992, 1994)

Revisions Include

- Fixing infelicities
- Real-time programming (protected records)
- Object oriented programming (single inheritance)
- Hierarchical library

Of concern for reliable computing

- More predictable semantics in the face of errors
- Safety and security annex
- Consideration of formal methods in the design
- Establishment of Rapporteur Group for critical software

Goals of the Language Precision Project

- Improve design and promote acceptance of Ada9X
  - Clarify problems
  - State and/or prove properties of the design
  - Identify verifiable subset
  - Provide basis for future formal work

Technical reports

- Phase 1: language issues
  - optimization
  - sharing and storage
  - protected records
  - overload resolution
  - the floating point model
  - distribution
  - program errors
  - object-oriented programming
- Phase 2: descriptive
  - "natural semantics" model
  - sequential dynamic semantics

This talk is an experience report

- Previous formal work on Ada
- Strategies
- Phase 1 case studies
- Phase 2 summary
- Conclusions

Previous formal work on Ada

- Formal definition projects
  - INRIA (official, little effect)
  - Dansk Datamatik (led to DDC compiler)
  - NYU (led to first validated compiler)
  - DDC/CRAI (almost whole language)
- Formal definitions of subsets, reasoning systems
  - Penelope (ORA)
  - AVA (CLInc)
  - Aerospace Corporation
  - SPARK (Program Validation Ltd.)
  - ProSpecTra
  - RAISE (Computer Resources International)

Phase 1: targets of opportunity

- Ignore well-understand parts of language
- Formalism and approach chosen "per problem"

Advantages

- Right level of abstraction
- Tractable: some hope of prompt response

Disadvantages

- Axiomatic method makes assumptions – how justified?
- Disregards interactions of features

Phase 2: Descriptive

- Describe (almost) final definition of sequential Ada 95
- Look for interactions between features
- Improve presentation of underlying conceptual model

Case studies

- Opportunism in action (default values)
- Overload resolution
- Object-oriented features

Opportunism in action: default values

- Proposal to permit default values for subtypes

```
subtype init is integer range 1 .. 100 := x+6;
y : init;       -- initialized to x+6;
z : init := 3;  -- initialized to 3;
```

- One goal, presumably is the following invariant:
  - Objects of subtype `init` will always have a defined value.

Example 1: Default values and out parameters

```
type init_rec is
  record
    ...
    comp : init;
  end record;

procedure Q(u : out init_rec) is
begin
  ...   -- Is u.comp initialized?
end Q;
```

Example 2: Default values and subtype conversions

```
subtype not_init is integer range 1..100;

type not_init_array is array(boolean) of not_init;
type init_array is array(boolean) of init;

a1 : not_init_array;
a2 : init_array := init_array(a1);
    -- Invariant fails for components of a2.
```


Model of defaults, elements

- Objects, statically associated with subtypes.
- States, associating values with objects.
- *Op*, a set of object-returning, side-effect-free operations.
- Execution consists of atomic acts creating objects and assigning values to them.

Model of defaults, terminology

- If x is an object
  - subtype(x) is the subtype of x
  - defaults(x) = all those subobjects y of x such that subtype(y) has a default value
- Good object (in a given state):
  - x is *good* iff every atomic object in defaults(x) has a defined value
  - The desired invariant says that all objects are good in all reachable states.
- A set of objects covers another object:
  - C covers x iff every object in defaults(x) is a subobject of some object in C

Model of defaults, execution

Execution is modeled by sequences of two atomic state-changing operations

- Assignment: `x := e`
  - Where e is an expression using operations from *Op*
- Creation and initialization: `create x: C := V`
  - Where C covers x and V assigns to the objects in C.

Model of defaults, example axioms

Axioms describe how state-changers affect the goodness of objects

- `y := e`
  - If y is a subobject of x, x is good, and e is good, then after this execution x is still good.
- `create x: C := V`
  - If the values assigned by V are good then, after this execution, x is good.
- Etc.

Immediate theorem:

The invariant is guaranteed if the operations in *Op* are non-decreasing: good arguments yield a good result under normal termination.

To apply the model:

Can the proposed language rules be described within the given model of defaults? If so, the invariant holds.

- Revisit example 1 (out parameters): How is an out parameter created?
- Revisit example 2 (subtype conversions): Is subtype conversion a non-decreasing operation?

Default subtypes, concluded

- Proposal dropped
- Not a "point change"

Overload resolution

- Ada83 rules are complex
  - Unpleasant surprises
  - Fine points not consistently interpreted by compilers
- Proposed changes add complexity (in some ways)
- Problems describing proposed changes
- Incompatibilities between old rules and proposed changes unclear

Problems

- Finding satisfactory rules for overload resolution
  - Interactions of features:
    - implicit conversion
    - OOP
    - strong typing
    - overload resolution
  - Special role of operator symbols
    - Proposal for "primitive visibility"
  - Blemishes in proposals
    - non-local
    - upward inconsistent
    - "Beaujolais effects"
- Describing the rules you've found
  - Descriptions of proposals (both declarative and algorithmic) flawed

Hierarchical names and "use" clauses

```
package p is
  function g return foo;
  function "+"(x,y: foo) return bar;
end package:
```

Compare

```
x : foo := P.g;
b : bar := P."+"(e1, e2);
```

with

```
use p;
x : foo := g;
b : bar := e1+e2;
```


Beaujolais effect

```
package P is
   function f(a: boolean) return boolean; -- F#1
end P;

function f(a: integer) return boolean;     -- F#2
function "<"(a: integer; b: integer) return integer;

[-- use P;]
x : boolean := f(1 < 2);    -- F#1 or F#2?
```


Abstract formal model of overload resolution

(by Bill Easton)

- Written in Z
- Parameters to model:
  - The environment (what declarations apply)
  - Which interpretations are preferred
- Hides irrelevant detail
  - Makes concrete syntax abstract (e.g., all expressions become applications)
  - Omits specs of irrelevant operations (e.g., the check that actuals match formals)
  - Applies to one expression at one point in program (environment is not a state variable)

Using the formal model of overload resolution

Different rules correspond to different choices of parameters

For particular rules, we may ask

- Are the rules "local"?
- Are there Beaujolais effects?
- Are there upward inconsistencies?
- What are the upward incompatibilities?

Results of work on overload resolution

- Understood
  - Formalism chosen to fit the problem
- Received with enthusiasm
  - Shed light on complex problem
- Suggested modifications adopted by MRT

Successful because

- Addresses acknowledged difficulty
- Suitable problem (isolatable)
- Right abstraction
- Right problem solver (mathematician and Ada lawyer)

Object oriented programming

Ada9X inheritance: generalizes type derivation

```
 with Calendar;
package alert_system is

   type alert is tagged
     record
       Time_Of_Arrival: Calendar.Time;
       Message: Text;
     end record;

   procedure Display(A: in Alert; ...);
   procedure Handle(A: in out alert);
   ...

   type medium_alert is new alert with
     record
       Action_Officer: Person;
     end record;

   ...

 end alert_system;
```


Classwide types:

The type Alert'class

- like a variant record type
- tag acts as discriminant
- alternatives are the descendants of Alert

Ada9X runtime binding:

```
procedure Process_Alerts(AC: in out Alert'Class) is
begin
   ...
   Handle(AC);    -- dispatch according to tag
   ...
end;
```


Goal of work on OOP:

- Proof formalism for a significant subset of the OOP constructs
- Extend existing formalism (Penelope) for Ada83

Interacting with the MRT

- Is the model right?
- Does the model cover enough?

Difficulties:

- OOP semantics is a research topic
- Ada9X OOP proposals very volatile
- Many non-semantic issues (syntax, static semantics, efficiency)
- LPT is an add-on

Conclusions of OOP report

- Proof formalism and specification notation for basic OOP mechanisms
- Justifies claim that semantics is clean
- Practical? Only applied to simple examples

Phase 2, approach

- Ignore static semantics
- Define dynamic semantics in "natural semantics" formalism
- Written in notation of Typed Prolog
  - static semantic checking
  - definition executable on small examples

Phase 2, summary

- Several problems found, corrected in final reference manual
- Surprises: difficulties in seemingly “trivial” areas

Conclusions: the Language Precision Project

- Opportunistic approach achieved some influence on the MRT
  - Some results unlikely to come from a different source
- Begun to identify a verifiable subset
- Limited by not being part of the MRT
  - Hard to find the best assignments
  - Not always in the loop
  - Contractual requirement for “products” not ideal
- Widely scattered LPT manageable, but awkward
- Differences in culture
  - LPT, semantics; MRT, implèmentation
  - Key demand on members of LPT – bridge the gap

Hope: work like this will become standard operating procedure