

Contract-based Programming A Route to Finding Bugs Earlier

Jacob Sparre Andersen

Jacob Sparre Andersen Research & Innovation

January 2015

Contract-based Programming

A software development technique, used to find programming errors earlier in the development process.

In its strictest form, the “contracts” are checked as a part of the compilation process, and only a program which can be proven to conform with the contracts will compile.

In a less strict form, it is more similar to “preventive debugging”, where the contracts are inserted as run-time checks, which makes it more likely to identify errors during testing.

In this presentation I will focus on how Ada 2012 supports contract-based programming, and give some guidelines on how to use the technique consistently.

Raison d'être for Ada

In the 1970's the US DoD noticed they had a problem with software development.

To solve it they arranged a programming language design competition with:

... three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency.

The result was Ada (1983).

(to make a long story short)

Ada in a single slide

- Procedural programming language.
- Supports object oriented programming (encapsulation, inheritance and dynamic dispatching).
- Modular programming: Packages, child packages and individual subprograms. Generic modules.
- Separates declarations of subprograms and packages in specifications and implementations.
- Concurrent, distributed and real-time programming built in.
- Types distinguished by name (both simple and composite types).

Most recent Ada standard published by ISO in December 2012.

Structure of the presentation

- **Types:**
A quick tour of the capabilities of the Ada type-system as it relates to contract-based programming.
- **Subprograms:**
An introduction to specifying the requirements and promises of subprograms in Ada.
- **Guidelines:**
Guidelines for applying contract-based programming consistently to types, subprograms and across whole packages/libraries.

Types

We can declare different **types** representing different kinds of values:

```
type Input_Voltage is delta 0.001 range -5.0 .. +5.0;

type Colours is (Red, Green, Blue);
type Months is (Jan, Feb, Mar, Apr, May, Jun,
                 Jul, Aug, Sep, Oct, Nov, Dec);

type Apples is range 0 .. 10_000_000;
type Oranges is range 0 .. 10_000_000;

type Disc_Point is private;
```

Types (continued)

The point of specifying a type is two-fold:

- To specify a collection of possible values and operations on the values. – For example integer values from 0 to 10'000'000 with the operations $+$, $-$, \times , $/$ and modulus.
- To separate different kinds of values. – For example to keep counts of apples and oranges separate, if that is intended.

Note that it is possible to have two separate types being able to represent exactly the same set of values.

Range constraints

In the previous examples we declared types each with some possible values.

If the type has a simple ordering, it may be possible to declare a subtype (subset) of the base type with more limiting upper and/or lower bounds on the possible values.

In the Ada standard library `Natural` is made a subset of `Integer` with a different lower bound:

```
subtype Natural is Integer range 0 .. Integer'Last;
```


Generalised constraints

We might want to put any, arbitrary constraints on which values are allowed in a subtype of a type.

Summer in the Southern hemisphere:

```
subtype Summer is Months  
  with Static_Predicate => Summer in Nov .. Dec |  
                                     Jan .. Apr;
```

Primes:

```
subtype Prime is Integer range 2 .. Integer'Last  
  with Dynamic_Predicate  
    => (for all N in 2 .. Prime - 1  
        => Prime mod N /= 0);
```

Generalised constraints (continued)

Make sure that `Disc_Point` objects stay on or inside the unit circle:

```
package Places is
  type Disc_Point is private;
  -- various operations on disc points
private
  type Disc_Point is
    record
      X, Y : Float range -1.0 .. +1.0;
    end record
  with Invariant => Disc_Point.X ** 2 +
                   Disc_Point.Y ** 2 <= 1.0;
end Places;
```

Adapted from the Ada 2012 rationale.

Types and type invariants (a kind of summary)

Contract-based programming is typically considered to be an extension of **strong, static typing**.

Contract-based programming extends the concept of types by allowing the programmer to declare “**subtypes**” whose values have to fulfill a constraint described in the form of an arbitrary boolean expression.

Globals and formal parameters

When we declare a subprogram, the first steps are to declare:

- Which formal parameters and global (state) variables are used and/or affected by the subprogram.
- If they are used and/or changed.
- The type of the formal parameters. (The type of the global variables must be declared elsewhere.)

```
procedure Increment (Counter : in out Integer;  
                    Step      : in      Integer);
```

```
function Voltage return Input_Voltages  
  with Globals => (Input => GPIO);
```

(the latter example is not checked by Ada compilers yet)

Subtypes for formal parameters

The next step is to narrow down the types of the formal parameters with subtypes where it is appropriate.

We count from zero and up (natural numbers). An increment is by one or more (positive numbers):

```
procedure Increment (Counter : in out Natural;  
                    Step      : in      Positive);
```

Preconditions for calling

In addition to specifying subsets of allowed values for formal parameters, we may have some conditions on the **system state** and **formal parameters** before it makes sense to call the subprogram.

These conditions are known as **preconditions**.

You can only write to open, writable files:

```
procedure Put (File : in      File_Type;
              Item : in      String)
  with Pre => (Is_Open (File)) and then
             (Mode (File) in Out_File | Append_File);
```

Preconditions for calling (continued)

Continuing our `Increment` example...

There is an upper limit (`Natural'Last`) to how far we can count with our selected type:

```
procedure Increment (Counter : in out Natural;  
                    Step      : in      Positive)  
with Pre => (Counter < Natural'Last);
```

We should not attempt an increment so large that we go beyond the upper limit of how far we can count (`Natural'Last`):

```
procedure Increment (Counter : in out Natural;  
                    Step      : in      Positive)  
with Pre => Counter <= Natural'Last - Step;
```

Postconditions of subprogram calls

The implementor of a subprogram may make certain promises (guarantees) about the **state of the system**, any **return values** and modified **formal parameters** once a subprogram returns.

These promises are known as **postconditions**.

The line number of a file is incremented when you write a line to it:

```
procedure Put_Line (File : in      File_Type;  
                   Item : in      String)  
  with Pre  => (Is_Open (File)) and then  
           (Mode (File) in Out_File | Append_File),  
  Post => (Line (File) = Line (File)'Old + 1);
```


Postconditions of subprogram calls (continued)

Continuing our `Increment` example...

Once we've incremented the counter it must be `Step` larger than the old value:

```
procedure Increment (Counter : in out Natural;  
                    Step      : in      Positive)  
with Pre  => Counter <= Natural'Last - Step,  
      Post => Counter = Counter'Old + Step;
```

Subprograms (a kind of summary)

Contract-based programming means that you:

- Specify which **global variables** are read and/or modified by each subprogram.
- Specify if global variables and formal parameters are **read from and/or written to**.
- Specify what each subprogram **requires** of the global system state and its formal parameters.
- Formalise your **promises** about the global system state, any return values and the values of the formal parameters as a subprogram returns.

Guidelines: Types, constraints and invariants

Make sure your type declaration is as detailed as possible.

- Declaring a new type or a subtype depends on what level of inter-type compatibility you want¹.
- Put an appropriate constraint on the range of values the (sub)type can have.
- Add any extra constraints as predicate (non-private types) or invariant (private types) aspects.

¹And if there is a type to derive from.

Example: Types, constraints and invariants

Primes are integers:

```
subtype Prime is Integer;
```

... larger than 1:

```
subtype Prime is Integer range 2 .. Integer'Last;
```

... and have no other factors than 1 and the prime itself:

```
subtype Prime is Integer range 2 .. Integer'Last  
with Dynamic_Predicate  
=> (not (for some N in 2 .. Prime - 1  
=> Prime mod N = 0));
```

Guidelines: Subprograms and argument declarations

Make sure that you declare the arguments for your subprograms as specifically as possible.

- Select the proper direction (“**in**”, “**out**” or “**in out**”) for each of the arguments to a subprogram.
- Select as specific a (sub)type as possible for each of the arguments to a subprogram.
- Use preconditions (postconditions) to declare stronger constraints on the input (output) values than those implied by the selected subtypes.

Example: Subprograms and argument declarations

We want to be able to increment a counter by arbitrary steps. We use (“in”) the value of both the counter and the step size to generate (“out”) a new value for the counter:

```
procedure Increment (Counter : in out Integer;  
                    Step      : in      Integer);
```

We count from zero and up (natural numbers). An increment is by one or more (positive numbers):

```
procedure Increment (Counter : in out Natural;  
                    Step      : in      Positive);
```

Example: Subprograms and argument declarations . . .

There is an upper limit (`Natural'Last`) to how far we can count with our selected type:

```
procedure Increment (Counter : in out Natural;  
                    Step      : in      Positive)  
with Pre => Counter <= Natural'Last - Step;
```

Once `Increment` returns `Counter` has changed:

```
procedure Increment (Counter : in out Natural;  
                    Step      : in      Positive)  
with Pre  => Counter <= Natural'Last - Step,  
      Post => Counter = Counter'Old + Step;
```

Guidelines: Subprogram preconditions

What are the requirements of your subprograms?

- Do some of your subprograms have some special requirements, which should be met before they can be called?
- Can a subprogram only be called once?
- Can a subprogram only be called when the system is in a specific state?

This can be documented with appropriately formulated preconditions to the subprograms.

Example: Subprogram preconditions

If we want to write to a file, it should be open and writable:

```
procedure Put (File : in      File_Type;  
              Item : in      String)  
  with Pre => (Is_Open (File)) and then  
            (Mode (File) in Out_File | Append_File);
```

Initialise only once:

```
procedure Initialise  
  with Pre => (State = Not_Initialised);
```

Packages

While we typically don't write contracts for entire packages, it does make sense to take a broader view of the pre- and postconditions of all the subprograms declared in a package.

If one specifies contracts one subprogram at a time, one may miss contract details on one subprogram, which would be helpful for another subprogram.

The following slides contain a few guidelines for ensuring consistent pre- and postconditions for entire packages.

Guidelines: Aligning pre- and postconditions

Do post- and preconditions match for likely sequences of calls to the declared subprograms?

- 1 Sketch likely, valid sequences of subprogram calls.
- 2 For each call in the identified sequences:
 - a Verify that the documented state of the input data matches constraints and preconditions for the called subprogram.
 - b If there is a mismatch: Attempt to narrow down the documented, possible output values of the source of the input data (by changing constraints and postconditions).
 - c Identify the documented state of the modified parameters after the call.

Example: Aligning pre- and postconditions

We look at a simple Text I/O package with some contracts added:

```
procedure Open (File : in out File_Type;  
                Mode : in      File_Mode;  
                Name : in      String);
```

```
procedure Close (File : in out File_Type);
```

```
procedure Put_Line (File : in      File_Type;  
                   Item : in      String)  
with Pre => (Is_Open (File)) and then  
        (Mode (File) in Out_File | Append_File),  
        Post => (Line (File) = Line (File)'Old + 1);
```

Example: Aligning pre- and postconditions

1 A likely call sequence:

```
Open      (File => Target,  
          Name => "output.txt",  
          Mode => Out_File);  
Put_Line (File => Target,  
          Item => "Hello.");  
Close    (File => Target);
```

2 • Open:

- a File, Name and Mode all ok. No preconditions.
- b (no mismatch)
- c Target can have any valid File_Type value.

Example: Aligning pre- and postconditions

- 2
 - Put_Line:
 - a Preconditions on `File` not matched by the documented constraints on `Target`. Item ok.
 - b `Target` was last modified by `Open`, so we add some appropriate postconditions there:

```
procedure Open (File : in out File_Type;  
               Mode : in      File_Mode;  
               Name : in      String)  
with Post => (Is_Open (File) and  
             Text_IO.Mode (File) =  
               Mode);
```

- 2
 - We now know that `Target` is open and has the mode `Out_File`.

Example: Aligning pre- and postconditions

- 2
 - Close:
 - a Close has no preconditions, so `Target` matches the **documented** requirements for the formal parameter `File`.
 - b (no mismatch)
 - c We know that `Target` has been changed, so it can have any valid `File_Type` value.

As some of you may have noticed, I have omitted to document that it is an error to open a file which already is open, or to close one which already is closed. – This is left as an exercise.

Compile-time checking of contracts

Static (compile-time) checking of contracts and aspects is currently only implemented, where it is required by the language standard – and in some cases for static subtype predicates.

In the future we will most likely see more cases of compilers or tools checking contracts already at compile-time, and not just inserting the checks in the running code.

Contact

Jacob Sparre Andersen
Jacob Sparre Andersen Research & Innovation
jacob@jacob-sparre.dk
<http://www.jacob-sparre.dk/>

Jacob Sparre Andersen

+45 21 49 08 04

Examples from this presentation:

[http://www.jacob-sparre.dk/reliability/
fosdem-2015-examples.zip](http://www.jacob-sparre.dk/reliability/fosdem-2015-examples.zip)

You can find my Open Source software repositories at:

<http://repositories.jacob-sparre.dk/>