

[print](#) | [close](#)

Use Contracts To Enforce Effective Coding

[Electronic Design](#)

[Hristian Kirtchev](#)

Hristian Kirtchev, AdaCore

Wed, 2014-03-05 09:34

As a software engineer, I always worry about code correctness. Have I met all the requirements in my code? Are any bugs lurking around? Back in college (think 1999) I was taught that the answers to these questions lie in testing. My debugging skills back then were limited to print statements. It sounds pretty frightening, but the college mentality was to “make it work,” rather than to produce correct code.

Related Articles

- [Strong Typing Yields Strong Programming](#)
- [Contract-Driven Programming Takes Specification Beyond The Stone Age](#)
- [C++ And Ada 2012: Renaissance Of Native Languages?](#)

After I graduated in 2005, I joined AdaCore as a compiler engineer. Working at the level of language semantics has given me insight into the strengths and weaknesses of various programming languages, in particular how the choice of language can affect the correctness and quality of a program. I have always liked strongly typed and robust languages, in particular Ada. I have been exposed to the usual suspects: C, C++, Java, and Python. But I always go back to Ada because I feel it is the safest.

Ada Today

Every so often, the Ada language is updated by a group of experts—a working group under the International Organization for Standardization (ISO)—who maintain the language based on technology directions and user experience. The latest version of the standard, Ada 2012, took a major step toward the integration of formal verification facilities into the core of the language. What I would like to

share with you is one of these facilities, called “contracts.”

The Merriam-Webster dictionary defines “contract” as “a binding agreement between two or more persons or parties; especially one legally enforceable.” Note two very important points. First, the agreement is binding. Both parties agree and accept the terms and ramifications of the contract. Second, the agreement is enforced by law.

Ada 2012 takes the concept of “terms and ramifications” and applies it at the subprogram level in the form of special annotations. To illustrate the use of these contractual annotations, I will use the classic stack operation “pop” as an example:

```
function Pop (S : in out Stack) return Element;
```

Popping from an empty stack is always considered a programming error. To verify this assumption, I would usually write some defensive code such as:

```
function Pop (S : in out Stack) return Element is
begin
  if Size (S) = 0 then
    raise Program_Error;
  end if;
  ...
end Pop;
```

or, equivalently:

```
function Pop (S : in out Stack) return Element is
begin
  pragma Assert (Size (S) > 0);
  ...
end Pop;
```

Note that the test for emptiness is the first statement executed within the implementation of Pop. In other words, it is a prerequisite for the correct behavior of Pop. Using Ada 2012 contracts, I would express the test in the following manner:

```
function Pop (S : in out Stack) return Element
with Pre => Size (S) > 0;
```

The “Pre” annotation (an “aspect” in Ada 2012 parlance) stands for “precondition.” The bit following the arrow => is a Boolean expression that must evaluate to True upon entry into routine Pop. The precondition sets the “terms” of the contract and announces to any potential caller that Pop expects a non-empty stack to operate properly. If this requirement is not met, the evaluation of the precondition raises an exception at runtime.

With the above I took care of the prerequisite. But what about the aftermath of calling Pop? I would like to somehow express the fact that the stack loses its top element after invocation. Enter postconditions:

```
function Pop (S : in out Stack) return Element
with Pre => Size (S) /= 0,
  Post => Size (S) = Size (S'Old) - 1
```

and then Pop'Result = Top (S'Old);

“Post” denotes a “postcondition.” The construct following the arrow => is a Boolean expression that must evaluate to True upon exit from routine Pop. The postcondition dictates the “consequence” of the contract and announces to any potential caller that the stack will always lose and return its topmost element.

Since the postcondition is more complex, I will break it down for you. The first part takes care of the size requirement of Pop:

Size (S) = Size (S'Old) - 1

(1) (2)

The size of stack S on exit from Pop (1) must be one less than the size of S upon entry into Pop (2). In other words, popping the stack will always cause it to shrink. Attribute 'Old is a nifty way of denoting the value of S before calling Pop. The next part of the postcondition:

Pop'Result = Top (S'Old)

(3) (4)

guarantees that the element returned by Pop (3) was indeed the top-most element of the stack before Pop was invoked (4). As you may have already guessed, attribute 'Result denotes the result of function Pop.

The careful reader may note that the postcondition is not as strong as it could or should be. For example, if the implementation of Pop shuffled all of the elements in the stack except for the topmost one, it would comply with the postcondition but violate the last-in first-out protocol that we associate with stacks. It is possible to express a more complete postcondition that captures the full functional requirements for Pop. That would involve the use of quantified expressions, another feature introduced in Ada 2012, and is somewhat outside the scope of this article.

As one of the engineers who worked on the implementation of attribute 'Old, I should confess that this Ada 2012 feature is potentially expensive. Internally, 'Old is transformed into a constant declaration that captures the value of its prefix. It may sound harmless, but in my example with Pop, 'Old ends up copying the whole stack! Using contracts during development and testing is a must. But I would like to somehow remove all this overhead when producing development code after I have verified that the program will never violate a precondition or postcondition, as it affects performance and memory consumption.

Ada 2012 offers a new pragma called “Assertion_Policy.” Instead of going through potentially thousands of lines of code and commenting out all contracts, I can simply write:

pragma Assertion_Policy (Pre => Ignore, Post => Ignore);

and be done with it!

The Value Of Contracts

From a software design point of view, contracts are a way of modeling requirements. Statements such as “popping an empty stack shall raise Program_Error” translate directly into a contractual annotation. This in turn greatly facilitates the process of code certification because the requirements are already synthesized in a compact and clearly visible manner in a

subprogram's specification rather than being spread out in implementation code.

Speaking of implementation, you may have already noticed that the contract of function Pop is completely independent of its implementation. I do not need to know whether Ms. Ada Programmer will use an array, a linked list, or some other data structure to represent the stack, yet I have the full power to express the desired constraints of its behavior through the use of contracts.

Once Ms. Programmer provides an implementation of my stack application programming interface (API), the contract I wrote takes the role of a runtime assertion check. Remember the "enforced by law" bit from the definition of contract? Here it is in action. The contract will detect any attempt to violate the requirements at runtime by raising an exception. This way the contract acts as a cutoff point and guards against bad input, bad output, or bad implementation. If Pop causes an issue during testing, all the quality assurance (QA) engineer has to do is set a breakpoint on the failed contract to get details on the violation. I find this to be a much better alternative than stepping through potentially an avalanche of code.

Contracts also have the added value of documentation. My colleagues at AdaCore come from eight different nationalities, and with different native languages we certainly have occasional issues of miscommunication and misinterpretation. Since contracts are written in standard Ada syntax, they act as unambiguous descriptions of requirements and behavior.

Contracts are a great foundation for safe programming and code correctness. I hope you too are considering what contracts can do for you and your development experience.

Hristian Kirtchev joined AdaCore in 2005 as a software engineer, but first experienced the company as an intern the year before. He is involved in the GNAT Pro front end, the implementation and integration of Ada 2005 language features, runtime support, testing, and quality assurance. He holds a BS and MS in computer science from the George Washington University and New York University, respectively. Outside of AdaCore, he is an avid pool and snooker player and a longtime gamer.

Source URL: <http://electronicdesign.com/embedded/use-contracts-enforce-effective-coding>