**Gem #65: gprbuild**

**Author:** Emmanuel Briot, AdaCore

**Abstract:** gprbuild is a new program builder, superseding gnatmake. It supports multiple languages, automatically manages source dependencies, and reduces the need for recompilation. This Gem describes a high-level view of how gprbuild works.

**Let's get started…**

GNAT allows for a great deal of flexibility in compiling applications. The traditional method is to use gnatmake on the command line, specifying the list of source directories by using "-I" switches, and letting gnatmake decide which files need to be recompiled. One drawback of this approach is that it only works for Ada files. If your application uses sources written in other languages (which is probably the case for the majority of serious applications), then you need some kind of wrapper around gnatmake. This is typically done via a Makefile: first you recompile C, C++, and other source files using standard Makefile techniques, then invoke gnatmake to handle the Ada sources, and finally (although this can also be done as part of gnatmake) bind and link your application.

Gnatmake is what we call a builder. Its role is to examine all source files in your application and decide which ones need to be recompiled. It then relies on other tools to do the actual compilation (gcc), bind (gnatbind) and link (gnatlink).

A few releases ago, a new tool called gprbuild was introduced in the GNAT technology. It is also a builder, reusing and extending gnatmake to support multilanguage applications. This means it knows how to handle C, C++, and many other languages, and is able to determine which source files for each language need to be recompiled. One benefit is that you no longer need to depend on a Makefile to compile those source files prior to binding and linking.

Gprbuild has been carefully designed so that no hard coding was done for any of the languages. Instead, all the information resides in a configuration file (typically having the extension ".cgpr"): how to compile a file, how to decide whether it is up to date, what files get generated as a result of the compilation, and so on.

These configuration files have a syntax similar to that of the project files themselves. They can be hand-edited as required, and maintained in your version control system as for any other component of the environment.

However, developing a configuration file is not always easy, especially when you are using different compiler chains (such as GNAT for Ada files and Diab for C). In particular, the link phase becomes harder to describe. Add to that support for multiple platforms, cross-compilation, libraries, and other subtleties, and the complexity increases quickly.

AdaCore has already taken care of a lot of possible compilers and combinations, so that you do not have to redo that complex setup yourself. We have created a knowledge base that describes the configuration for various common scenarios.

GNAT comes with an additional command-line tool, called gprconfig, which is generally spawned transparently by gprbuild, so you don't have to invoke it directly in most cases. The role of gprconfig is to create the configuration file based on your specific set of compilers, getting information from the knowledge base.

Now that we've described the general organization of things, let's show a concrete example.

Let's assume your application is made up of two files, bar.adb and foo.c, with the following code (not very interesting, admittedly, since it will do exactly nothing):

```
procedure Bar is
   procedure Foo;
   pragma Import (C, Foo);
begin
   Foo;
end Bar;
```

```
int foo () {}
```

gprbuild only works with project files (there are no command-line options to specify source directories, for instance). So you need to write one. The basic elements of information you need to specify in a project file are: languages (C and Ada in our case), source directories and source files (implicit in our case), and main source files (bar.adb in our case). So the project file default.gpr looks like:

```
project Default is
   for Languages use ("C", "Ada");
   for Main use ("bar.adb");
end Default;
```

At this point everything is set up, and we can just spawn gprbuild with the command:

```
gprbuild -Pdefault
```

Since we did not specify a configuration file through a –config switch, gprbuild will automatically generate one by spawning gprconfig and telling it we want to find the first available Ada and C compilers on the PATH. Gprconfig will then create a configuration file called auto.cgpr, which gprbuild in turn uses to build our executable.

If you relaunch the same command again immediately, no recompilation takes place, not even for the C file.

Notice also how gprconfig selects the first matching compiler on the PATH for each language. If you want to force the use of specific compilers (that might not even be on

your PATH), you will need to spawn gprconfig manually yourself and pass it a –config switch. This creates a configuration file which you can then pass along on your invocation of gprbuild.

**Gem #66: GPS's Key Shortcuts Editor**

**Author:** Emmanuel Briot, AdaCore

**Abstract:** Most GPS features are accessible through menus and contextual menus. However, for maximum efficiency most users prefer to use keyboard shortcuts. Although GPS comes with a number of predefined shortcuts, you might want to adapt them to your own habits, especially if you are moving from a previous editor.

**Let's get started…**

There are several ways in which the keyshortcuts can be changed in GPS.

One of the common cases is a user previously using the Emacs editor. GPS comes with a plug-in that will change the default key bindings in GPS. You just need to go to the /Tools/Plug-ins menu, and activate the "emacs.xml" plug-in. Note that for full compatibility with Emacs, you should also load some additional plug-ins (these are documented in the Description tab).

Similar plug-ins can be built for other editors, and shared with the rest of your team.

The most flexible method, however, is to go to the /Edit/Key Shortcuts menu. This will open a dialog in which you can configure shortcuts for any of the menus or any of the actions that GPS has created.

At this point, we should explain the difference between actions and menus. Internally, GPS exports its various features to the GUI as named actions. These actions are implemented either in Ada or using one of the scripting languages supported by GPS (python or XML for instance). They are organized into categories to make them easier to find. You can also create your own actions through XML files (see the GPS documentation for more details).

Most of the time, these actions are hidden and called in the background. But you can activate the "execute extended" plug-in if you want to be able to directly execute these commands by name, as Emacs does through its "M-x" key binding.

Some of these actions are then bound to menus or contextual menus.

Back to the Key Shortcuts editor. The main part of the dialog shows the list of actions exported by GPS. These are organized into categories. One of them is special, and is called "Menus". It gives you access to all the menus of GPS (so some actions are accessible in two places, either by category/name, or by the menu). If you activate the "Flat List" button, the actions will instead be listed alphabetically and the categories will no longer be visible.

Next to each action or menu, GPS displays the shortcut associated with that action. One limitation here is that the shortcut is either shown next to the menu or next to the action (depending on how it was setup), but not both.

Here's a small trick if you want to find out what a shortcut is bound to. You can click on both the "Flat List" and the "Shortcuts Only" buttons. This will list only those actions which currently have a shortcut associated. By clicking on the "Shortcut" column header, you can then sort the list by shortcut, thus making it easy to find out what a specific key shortcut does.

Through this dialog, you can change the shortcut for any of the actions. Simply select that action in the list, then press the "Grab" button, and you can then press the keyboard shortcut that you want to assign to it.

Key shortcuts are not limited to a single key (which on a standard keyboard would not provide enough shortcuts for those Emacs users). You can in fact have a series of shortcuts, as in "Control-x k", which is used in Emacs to close the current window. Due to limitations in the gtk+ toolkit on which GPS is based, however, the menus are only able to display single shortcuts, so if you assign the above shortcut to a menu you can use the shortcut but it will not be visible in the menu itself.

Once you close the dialog with "OK", the shortcut becomes effective immediately, and will be preserved from session to session. In fact, the key shortcuts you just created are saved in an XML file (keys.xml) in your $HOME/.gps directory. You can thus copy this file to other accounts or setups to automatically get the same shortcuts again.

**Gem #67: Managing the GPS Workspace**

**Author:** Emmanuel Briot, AdaCore

**Abstract:** GPS has a multitude of views and editors, several of which may be displayed on the screen at the same time. It is based on a very flexible desktop that helps you organize these windows the way you prefer. This Gem describes some of the lesser-known aspects of the GPS desktop.

**Let's get started…**

When you open GPS initially, it shows a number of views: the Messages window (which can never be closed), the Project View, showing the list of files in your project, the Outline View, which shows the list of subprograms in the current edit, the Scenario View listing the scenario variables in your project, and a Welcome window to help you get started.

When several windows share the same screen area (which is the case for the project view and the outline view in the default desktop), they are organized into notebooks, and it's possible to show any view by clicking on its associated tab. By default, GPS will now show the tabs if there is only one window in a given area, to save screen real estate. This behavior can be changed through the "Notebook Tabs Policy" preference. You can also change the location of the tabs (which by default appear below the window) through the "Notebook Tabs Position" preference. This latter preference configures the default location, though, and you can change it on a per-notebook basis by right-clicking in any of the tabs and changing the tabs position.

Windows can also be made to float. They are then put under the control of the operating system (or the window manager), and you can move them to another screen, another virtual desktop, iconify them, and so on. The window will have its own entry in your systems panel (the bar that is generally displayed at the bottom of the screen and shows all open windows). GPS has a mode where all windows it creates are made floating by default (this is the "All Floating" preference). When this mode is not activated, you can move a floating window back into GPS using one of two methods: the /Window/Floating menu, or by clicking on the [x] in the window's title bar when the "Destroy Floating" preference is unset; at that point, instead of closing the window, GPS will simply put it back in its own window. If you really want to close it, you have to click a second time on [x].

Only one of the views has the focus at any given time, and it receives all keyboard input. Its title bar is then highlighted in a different color (which you configure in the preferences). To save some space on the screen, you could choose to hide the title bars. Indeed, it is obvious most of the time what each window is after you have used them a few times. The only case where you might be missing info is for editors (since the title bar includes the full name of the file). However, this name also appears in GPS's own

title bar. When the title bar is hidden, the notebook's tab will be color highlighted to show the currently active window.

When a window is not currently visible on the screen (most often this will be when the Messages window is below the Locations window for instance), its tab is highlighted in a special color to bring attention to it and encourage you to check what has changed in that window.

It is rather obvious (or so we hope) that the views can be resized as you see fit simply by dragging the separators between those windows (that is, click on the separator, hold the mouse button, and move the mouse up/down or left/right, depending on the orientation of the separator). The resizing can take one of two forms: either a simple line overlaid on top of the windows to show where the separator will end up when you release the mouse, or by redrawing the windows each time you move the mouse. This behavior can also be configured in the preferences.

However, did you know that you can actually create any number of new areas on the desktop (which we call the MDI, for Multiple Document Interface)? This can be done in a number of ways. The easiest to discover is to use the menu /Window/Split Side-by-Side or /Window/Split Up-Down. For instance, assuming you have the default desktop loaded in GPS, select the project view. Then select the menu /Window/Split Up-Down. This will put the outline view in its own area below the project view, allowing you to see both at the same time.

However, the most flexible way to reorganize windows is by using drag and drop. Start by clicking on the title bar or the tab of any window. Then, without releasing the mouse button, move your mouse to where you would like to put the window you clicked on. Note how a pop-up window appears on top of the GPS window to describe what will happen when you release the mouse. There are six places where you can drop a window: if you release the mouse in the middle of an existing window, the two windows will be stacked into a notebook; if you release on any of the sides of an existing window (there is a margin a few pixels wide on each side), the existing window will be split and the window you are moving will be displayed next to it; finally, you can also release the mouse outside of the GPS window altogether to make the window floating.

As an additional bonus, if you press the shift key when you are dropping an editor window, a new view will be created (showing the same contents) instead of moving the initial editor. This allows you to view several locations in a given file, which is often useful when you want to view data structures when writing code, or when comparing several sections of code.

My own preferred organization (on a wide screen) is as follows: I have two editors side by side (for instance spec and body). Then on the left side I display the Windows view (/Tools/Views/Windows menu), which provides a fast way to select and bring to the front any window no matter where it currently is on the screen. This is a also a convenient place from which to start the drag-and-drop operations (clicking on a file and dropping it

where I want to put the editor). Below this Windows view I have the Bookmarks view, which allows me to save places within any of the editors (through the /Edit/Create Bookmark menu), and come back to them later. Below the two editors, I display the Messages window and the Locations window side by side, so that I can see compilation errors easily, while keeping an eye on whether GPS has reported an error in its messages window. Finally, on the right-hand side of the screen I display the Outline view, since this is such a great way to move fast within a file, and the Tasks view that shows what's being executed in the background (and provides a way to monitor the progress of compilations when using the -d switch to gnatmake). As you see, there are quite a number of views. To save space, I usually hide title bars (which as explained above are not necessary once you know what the windows are for), as well as the status bar at the bottom of the GPS window (since the task view displays the same information already).

Once you have found a layout that you like, however, you probably want to reuse it every time you open GPS. This is called saving the desktop. This process is very configurable, to adapt to the various needs that people might have:

– The default is for GPS to save the project-specific desktop. This means that the layout you have just created will be saved automatically on exit, and reapplied when loading the same project later on. If you load another project, the default layout will be used. One reason to do that is to save the currently opened files in the desktop (this is controlled by another preference, "Save Editor is Desktop", where you decide whether you want to save editors or not, and if you do, whether only project-specific editors will be saved or any file that happens to be opened at that time.

- If you want to reuse the same layout for all projects on which you work, you should make sure the preference "General/Save Project-Specific Desktop on Exit" is disabled. This means that a single version of the desktop will be saved, and will apply to all projects (in this case it might be a good idea to remove the file $HOME/desktop.xml prior to launching GPS the first time, so that any trace of project-specific desktop is wiped out). In such a case, one of the drawbacks is that you might not want to save the editors since they would not necessarily match the next project you load.

The method I have been using is the following: after removing the desktop.xml file as described above, I launch GPS and set up the various views as I like, but do not open any editor. I then save the current desktop as the default through the menu /File/Save Move/Default Desktop. I then ensure that the preference to save project-specific desktops is on, as well as the preference to save the editors in the desktop. This ensures that every time I open a project that I've never opened before, I get the default desktop (with no editor), but when I open a project that I have already opened before, then I get the same editors (and in the same location) as the last time I exited GPS.

**Gem #68: Let's SPARK! – Part 1**

**Author:** Yannick Moy, AdaCore

**Abstract:** In this Gem and the next one, we present a simple walk-through of SPARK's capabilities and its integration with GPS. In this first Gem, we show how to set up a SPARK project and prove that your SPARK programs are free from uninitialized variable accesses and that they execute without run-time errors.

**Let's get started…**

With Praxis and AdaCore now teaming up to offer an integration of SPARK technology inside GPS (see http://www.adacore.com/home/products/sparkpro/), many GPS users will be interested in trying out the proof capabilities of SPARK on their own Ada programs. Of course it's a little more involved than that. SPARK is not only a set of tools for verifying high-assurance systems, but also incorporates a language that must be learned.

The SPARK language is made up of two parts, one of which is a subset of Ada (whose features will obviously already be familiar to Ada programmers!), meant to facilitate code understanding and proofs, and the other part being a specification language, meant to express properties of programs. Quite conveniently, the SPARK specifications (a.k.a. SPARK annnotations, to distinguish them from Ada specifications) are expressed within stylized Ada comments of the following form:

```
--# <some annotation here>
```

so SPARK annotations don't interfere with normal Ada compilation.

This Gem and the next one demonstrate various of the properties you can prove with SPARK, and how these relate to the SPARK annotations written by the user. As a simple example, we will take a procedure which searches linearly for a value in an array, and returns the index, if any, at which the value is found.

Here is the specification file search.ads:

```
package Search is

  type IntArray is array (Integer range <>) of Integer;

  procedure Linear_Search
    (Table : in IntArray;
     Value : in Integer;
     Found : out Boolean;
     Index : out Integer);

end Search;
```

And the body file search.adb:

```
package body Search is

  procedure Linear_Search
    (Table : in IntArray;
     Value : in Integer;
     Found : out Boolean;
     Index : out Integer)
  is
     I : Integer := 0;
  begin
     Found := False;

     while I <= Table'Last loop
        if Table(I) = Value then
           Found := True;
           Index := I;
           exit;
        end if;
        I := I + 1;
     end loop;
  end Linear_Search;

end Search;
```

Let's first get set up for using SPARK:

1. Copy the files search.ads and search.adb into a directory named search.
2. Open GPS with a default project in the same directory.
3. Expand file search.adb.
4. Select SPARK/SPARKMake from the menu. This generates a file search.idx.
5. Copy the following code to a file called search.cfg:

```
  package Standard is
     type Integer is range -2**31 .. 2**31-1;
  end Standard;
```

6. From the Menu, select Project/Edit Project Properties.
7. Go to the page Switches/Examiner.
8. Select the following options:

```
  Index File            : search.idx
  Configuration File    : search.cfg
  Analysis              : Data Flow only
  Generate VCs          : yes (check it)
```

That's it!

Now open search.adb and select SPARK/Examine File. The SPARK Examiner runs and outputs the following messages:

```
       Flow Error 602 - The undefined initial value of Index may be
                        used in the derivation of Index
```

```
Warning 402 - Default assertion planted to cut the loop
Note - Information flow analysis not carried out
```

The Flow Error indicates that, although Index is an out parameter, it is not initialized on all paths through Linear_Search. One could argue that our intent here is to access Index only when Found is set to True. SPARK considers initialization errors too serious to allow such subtleties, and requires that all paths through the procedure must initialize all out parameters. Let's comply and initialize Index:

```
begin
   Found := False;
   Index := 0;
```

After we rerun the Examiner, we are left with a Note that is simply a reminder of our choice of options, and a Warning that we will explain in the next Gem.

Now, we can continue with proving that our procedure is free from run-time errors, such as integer overflow and out-of-bounds array accesses. Select SPARK/Simplify All. The SPARK Simplifier runs. To see the result of this tool's execution, select SPARK/POGS. This opens a file search.sum, which summarizes all the proofs in the following table:

VCs for procedure_linear_search :

| # | From | To | Proved In: vcg | Proved In: siv | Proved In: plg | Proved In: prv | False | TO DO |
|---|---|---|---|---|---|---|---|---|
| 1 | start | rtc check @ 13 | | YES | | | | |
| 2 | start | assert @ 15 | | YES | | | | |
| 3 | 15 | assert @ 15 | | YES | | | | |
| 4 | 15 | rtc check @ 16 | | | | | | YES |
| 5 | 15 | rtc check @ 18 | | YES | | | | |
| 6 | 15 | rtc check @ 21 | | | | | | YES |
| 7 | 15 | assert @ finish | YES | | | | | |
| 8 | 15 | assert @ finish | YES | | | | | |

Each line corresponds to a Verification Condition (VC), which must be proved in order to guarantee that the program is free from run-time errors. Each column corresponds to the result of the proof attempt. If YES appears in one of the 4 "Proved In" columns, the proof was successful. If YES appears in the "False" column, there is something definitely wrong. If "YES" appears in the "TO DO" column, we don't know: either the program is wrong, or it is too complex to prove.

Since column "TO DO" is not empty here, not all proofs were successful. In the next Gem, we will give you more details about failed proofs. The reason for the failures here is that program Linear_Search is incorrect, meaning that run-time errors can be raised. To see this, just complete the program with the following code in main.adb, and build the executable.

```
with Search;
use Search;
```

```
procedure Main is
   Table : IntArray(1..10) := (others => 0);
   Found : Boolean;
   Index : Integer;
begin
   Linear_Search(Table, 0, Found, Index);
end Main;
```

Now run the executable, and it raises an exception:

```
   raised CONSTRAINT_ERROR : search.adb:16 index check failed
```

This is because we are passing an array to Linear_Search that starts at index 1 in its parameter Table, whereas Linear_Search loop assumes that the array starts at index 0. SPARK correctly assumes that we can pass in such an array to Linear_Search, and thus it fails to prove that Linear_Search is free from run-time errors.

Let's correct the code of Linear_Search:

```
  procedure Linear_Search
    (Table : in IntArray;
     Value : in Integer;
     Found : out Boolean;
     Index : out Integer) is
  begin
     Found := False;
     Index := 0;

     for I in Integer range Table'Range loop
        if Table(I) = Value then
           Found := True;
           Index := I;
            exit;
        end if;
     end loop;
  end Linear_Search;
```

Let's do it again: Examine, Simplify All, POGS …
This time, columns "False" and "TO DO" are empty, meaning that all proofs were successful.

```
VCs for procedure_linear_search :
------------------------------------------------------------------------------
     |       |                     |  -----Proved In-----  |       |       |
#    | From  | To                  | vcg | siv | plg | prv | False | TO DO |
------------------------------------------------------------------------------
1    | start | rtc check @ 11      |     | YES |     |     |       |       |
2    | start | rtc check @ 13      |     | YES |     |     |       |       |
3    | start |    assert @ 13      |     | YES |     |     |       |       |
4    | 13    |    assert @ 13      |     | YES |     |     |       |       |
5    | 13    |    assert @ 13      |     | YES |     |     |       |       |
6    | 13    | rtc check @ 14      |     | YES |     |     |       |       |
7    | 13    | rtc check @ 16      |     | YES |     |     |       |       |
```

```
8    | 13    |    assert @ finish  | YES |     |     |     |       |       |
9    | 13    |    assert @ finish  | YES |     |     |     |       |       |
-------------------------------------------------------------------------------
```

Thus, we have proved that procedure Linear_Search is free from run-time errors.

In the next Gem, after the summer break, we will prove that procedure Linear_Search actually respects a given contract.

### Related Source Code

**Gem #69: Let's SPARK! – Part 2**

**Author:** Yannick Moy, AdaCore

**Abstract:** In this Gem and the previous one, we give you a simple walkthrough of SPARK's capabilities and its integration with GPS. In the previous Gem, we showed how to set up a SPARK project and prove that your SPARK programs are free from uninitialized variable accesses and that they execute without run-time errors. In this Gem, we show how to prove that your SPARK programs respect given contracts.

**Let's get started…**

In the last Gem, we proved that procedure Linear_Search was free from uninitialized variable accesses and run-time errors, which are safety properties of Linear_Search.

Now we can try to prove a specific behavioral property of Linear_Search, expressed as a contract between Linear_Search and its callers. A contract will consist of a precondition that callers of Linear_Search are responsible for establishing, before calling Linear_Search, and a postcondition that Linear_Search must establish, before returning to the caller. If not present, a default "true" pre- or postcondition is assumed.

Let's prove that when Linear_Search returns with Found = True, the value of Table at Index is Value. This can be expressed in SPARK as a postcondition on procedure Linear_Search that is located after its declaration in search.ads:

```
  procedure Linear_Search
    (Table : in IntArray;
     Value : in Integer;
     Found : out Boolean;
     Index : out Integer);
  --# post Found -> Table(Index) = Value;
```

Notice the implication symbol ->, which is only defined in SPARK annotations.

Let's call the SPARK tools, as we did in the previous Gem: Examine, Simplify All, POGS …
This time, the column "TO DO" is not empty:

```
VCs for procedure_linear_search :
------------------------------------------------------------------------------
     |       |                      |  -----Proved In-----  |       |       |
#    | From  | To                   | vcg | siv | plg | prv | False | TO DO |
------------------------------------------------------------------------------
1    | start | rtc check @ 11       |     | YES |     |     |       |       |
2    | start | rtc check @ 13       |     | YES |     |     |       |       |
3    | start |    assert @ 13       |     | YES |     |     |       |       |
4    | 13    |    assert @ 13       |     | YES |     |     |       |       |
5    | 13    | rtc check @ 14       |     | YES |     |     |       |       |
6    | 13    | rtc check @ 16       |     | YES |     |     |       |       |
7    | 13    |    assert @ finish   |     | YES |     |     |       |       |
8    | 13    |    assert @ finish   |     |     |     |     |       |  YES  |
```

```
------------------------------------------------------------------------
```

If we right-click on this line and select SPARK/Show Simplified VC, GPS opens a file linear_search.siv, which shows that the unproved VC corresponds precisely to the postcondition we just added:

```
C1:    found -> element(table, [index]) = value .
```

This is the conclusion (C) that the prover tries to prove with the set of hypotheses (H) above it. If we look at the hypotheses, we see that the conclusion cannot indeed be proved. This has to do with the warning which we already saw in the previous Gem:

```
Warning 402 - Default assertion planted to cut the loop
```

To prove a property of a procedure with a loop, we cannot unroll the loop an unbounded number of times. Therefore, SPARK "cuts" the loop with a loop invariant, which is a property that the loop maintains. Unless you provide such a loop invariant, SPARK assumes by default that nothing is maintained through the loop. If you provide one, SPARK will prove three things:

1) the loop invariant holds when control enters the loop for the first time
2) the loop invariant is maintained during an arbitrary run through the loop
3) the loop invariant is sufficient to prove the postcondition

What is missing in our case is the information that Found remains False throughout the loop. Let's add it, with the following syntax in search.adb:

```
for I in Integer range Table'Range loop
   --# assert Found = False;
```

Let's do it again: Examine, Simplify All, POGS …
Notice that the warning about a default assertion disappeared.
This time, the "TO DO" column is empty, so we have successfully proved Linear_Search's postcondition!

Finally, let's see how SPARK deals with global variables, by adding a counter to Linear_Search, which is incremented by one each time the call succeeds. We need to state explicitly that Counter is part of the state of this package, which we do using an "own" annotation below. We also need to state explicitly that Counter is initialized using an "initializes" annotation. The following declaration should go into search.ads:

```
package Search
--# own Counter;
--# initializes Counter;
is
   Counter : Natural := 0;
```

Now, we increment Counter in Linear_Search's body in search.adb:

```
      if Table(I) = Value then
         Counter := Counter + 1;
```

If we try to run the Examiner at this point, it flags an error:

```
Semantic Error 1 - The identifier Counter is either undeclared or not
                   visible at this point
```

This is because reads and writes of global variables are part of a subprogram specification in SPARK. Since Linear_Search does not declare in its specification that it reads or writes a global variable, it is not allowed to do so in its body. So let's add a SPARK annotation to state that Linear_Search reads and writes Counter.

```
procedure Linear_Search
  (Table : in IntArray;
   Value : in Integer;
   Found : out Boolean;
   Index : out Integer);
--# global in out Counter;
--# post Found -> Table(Index) = Value;
```

Let's do it again: Examine, Simplify All, POGS …
We get a new unproved VC in the column "TO DO", which corresponds to the following conclusion:

```
C1:    counter <= 2147483646 .
```

This time, it is because SPARK has detected a problem during the proof that the update of Counter does not overflow. Indeed, it could overflow! The solution is to add a precondition to Linear_Search, that promises that it will never be called in a state where Counter is the largest integer value.

```
procedure Linear_Search
  (Table : in IntArray;
   Value : in Integer;
   Found : out Boolean;
   Index : out Integer);
--# global in out Counter;
--# pre  Counter < Integer'Last;
--# post Found -> Table(Index) = Value;
```

As before, we must modify the loop invariant. Here, we just have to repeat this information in the loop invariant:

```
    for I in Integer range Table'Range loop
       --# assert Found = False and Counter < Integer'Last;
```

Let's do it again: Examine, Simplify All, POGS …
Everything is proved!

Notice that the promise made by the precondition will have to be proved by callers of Linear_Search …

Finally, let's add to the SPARK annotation of Linear_Search that Counter is incremented by one when Linear_Search returns Found = True. This can be expressed as a postcondition relating the value of Counter at procedure entry, denoted Counter~, and the value of Counter at procedure exit, denoted Counter:

```
procedure Linear_Search
  (Table : in IntArray;
   Value : in Integer;
   Found : out Boolean;
   Index : out Integer);
--# global in out Counter;
--# pre  Counter < Integer'Last;
--# post Found -> (Table(Index) = Value and Counter = Counter~ + 1);
```

Notice that in the precondition we simply use Counter to denote the value at procedure entry, because the precondition is precisely evaluated at procedure entry.

As before, we update the loop invariant to state that the current value of Counter is the same as the one at procedure entry:

```
    for I in Integer range Table'Range loop
       --# assert Found = False and Counter < Integer'Last
       --#        and Counter = Counter~;
```

Let's do it a last time: Examine, Simplify All, POGS …
Everything is proved!

Remember though that we can only prove a contract we wrote, which may be very different from saying abstractly that procedure Linear_Search is "correct". Who knows what the correct behaviour of Linear_Search means for the human who programmed it?

Let's recap what we have seen so far. SPARK is a language that combines a strict subset of Ada with annotations written inside stylized Ada comments. We have seen various kinds of SPARK annotations: preconditions introduced by pre; postconditions introduced by post; loop invariants introduced by assert; frame conditions introduced by global, own, and initializes. The SPARK tools allow us to check that a procedure is free from uninitialized variable accesses, that it executes without run-time errors, and that it respects a given contract, written next to its declaration, that callers can rely on.

In later Gems we will explore in more depth the capabilities of SPARK and its integration with GPS. In the meantime, you can learn more about SPARK in the SPARK tutorial at http://www.adacore.com/home/products/sparkpro/tokeneer/discovery/.

**Gem #70: The Scope Locks Idiom**

**Author:** Pat Rogers, AdaCore

**Abstract:** This Gem marks a brief break from SPARK, which will return in two weeks with the first in a series of six Gems on that topic. Encapsulating shared variables inside protected operations is not always possible. This Gem shows how to add mutual exclusion to existing sequential code using a combination of controlled and protected types, such that the resulting code is robust and minimally changed.

**Let's get started…**

Like the classical "monitor" concept on which they are based, protected types provide mutually exclusive access to internal variables. Clients can only access these variables indirectly, by means of a procedural interface. This interface is very robust because mutually exclusive access is provided automatically: users cannot forget to acquire the underlying (logical) lock and cannot forget to release it, including when exceptions occur. As a result, encapsulating actions within protected operations is highly recommended.

However, applying a protected type and protected operations may not always be feasible. For example, consider an existing sequential program that makes calls to procedures and functions provided by a package. Inside the package are variables that are manipulated by the procedures and functions. If more than one task is now going to be calling these subprograms, the package-level variables will be subject to race conditions because they are (indirectly) shared among the calling tasks. Moving the procedures and functions into a protected object would provide the required mutual exclusion, but it would require changes to both the package and the callers. Additionally, the existing procedures and functions may perform potentially blocking operations, such as I/O, that are prohibited from within protected operations.

In such a case, the programmer must fall back to manually acquiring and releasing an explicit lock. The result is essentially that of using semaphores, a low-level and clearly much less robust approach. For example, to ensure serial execution of the exported operations, one could declare a lock at the package level, as shown below, and have each operation acquire and release it:

```
...
package body P is

   Mutex : Mutual_Exclusion;

   State : Integer := 0;

   procedure Operation_1 is
   begin
      Mutex.Seize;
      State := State + 1;  -- for example...
      Put_Line ("State is now" & State'Img);
      Mutex.Release;
```

```
   exception
      when others =>
         Mutex.Release;
         raise;
   end Operation_1;

   procedure Operation_2 is
   begin
      Mutex.Seize;
      State := State - 1;  -- for example...
      Put_Line ("State is now" & State'Img);
      Mutex.Release;
   exception
      when others =>
         Mutex.Release;
         raise;
   end Operation_2;

end P;
```

The type Mutual_Exclusion is actually a subtype of a binary semaphore abstraction available to users via the GNAT.Semaphores package:

```
  subtype Mutual_Exclusion is Binary_Semaphore
    (Initially_Available => True,
     Ceiling             => Default_Ceiling);
```

See package GNAT.Semaphores for the details. You can assume it is a protected type with classic semaphore semantics. We define a subtype to ensure that all such objects are initially available, as required when providing mutual exclusion.

Although we cannot eliminate the need for this lock, we can make the code more robust by automatically acquiring and releasing it using an object of a controlled type. Initialization will automatically acquire the lock and finalization will automatically release it, including both when an exception is raised and when the task is aborted. (C++ programmers may be familiar with this technique under the name “Resource Acquisition Is Initialization” (RAII).)

The idea is to define a limited controlled type that references a shared lock using a discriminant. Objects of the type are then declared within procedures and functions with a discriminant value designating the shared lock declared within the package. Such a type is called a “scope lock” because the elaboration of the enclosing declarative region – the scope – is sufficient to acquire the referenced lock. The subprogram’s sequence of statements will not execute until the lock is acquired, no matter how long that takes. When the procedure or function is done, for any reason, finalization will release the lock. The resulting user code is thus almost unchanged from the original sequential code:

```
...
package body P is

   Mutex : aliased Mutual_Exclusion;

   State : Integer := 0;

   procedure Operation_1 is
      S : Scope_Lock (Mutex'Access);
   begin
      State := State + 1;  -- for example...
      Put_Line ("State is now" & State'Img);
   end Operation_1;

   procedure Operation_2 is
      S : Scope_Lock (Mutex'Access);
   begin
      State := State - 1;  -- for example...
      Put_Line ("State is now" & State'Img);
   end Operation_2;

end P;
```

To define the Scope_Lock type, we declare it with a discriminant designating a Mutual_Exclusion object:

```
  type Scope_Lock (Lock : access Mutual_Exclusion) is
     tagged limited private;
```

In the private part the type is fully declared as a controlled type derived from Ada.Finalization.Limited_Controlled, as shown below. We hide the fact that the type will be controlled because Initialize and Finalize are never intended to be called manually.

```
   type Scope_Lock (Lock : access Mutual_Exclusion) is
      new Ada.Finalization.Limited_Controlled with null record;

   overriding procedure Initialize (This : in out Scope_Lock);
   overriding procedure Finalize   (This : in out Scope_Lock);
```

Each overridden procedure simply references the semaphore object designated by the formal parameter's discriminant:

```
  procedure Initialize (This : in out Scope_Lock) is
  begin
    This.Lock.Seize;
  end Initialize;

  procedure Finalize (This : in out Scope_Lock) is
  begin
    This.Lock.Release;
  end Finalize;
```

So as you can see, by combining controlled types with protected types one can make simple work of providing mutually exclusive access when a protected object is not an option. By taking advantage of the automatic calls to Initialize and Finalize, the resulting user code is much more robust and requires very little change.

## Related Source Code

**Gem #71: Tokeneer Discovery – Lesson 1**

**Author:** Dean Kuo and Angela Wallenburg, Praxis High Integrity Systems

**Abstract:** In previous Gems, we saw how to create a SPARK project in GPS, and how to run the Examiner and Simplifier tools to verify various properties of a simple linear search function. In this series of Gems on SPARK, we will explore SPARK capabilities in greater depth, using source code from Tokeneer, an NSA-funded, highly secure biometric software system. In this Gem, we show how to deal with improper initialization.

**Let's get started…**

Note: This series of Gems makes references to SPARK and the Tokeneer project, which you can find out more about by visiting http://www.adacore.com/home/products/sparkpro/tokeneer/. See also the SPARK tutorial at http://www.adacore.com/home/products/sparkpro/tokeneer/discovery/.

As stated in the SANS Common Weakness Enumeration (http://cwe.mitre.org/top25/#CWE-665): "If you don't properly initialize your data and variables, an attacker might be able to do the initialization for you, or extract sensitive information that remains from previous sessions. When those variables are used in security-critical operations, such as making an authentication decision, then they could be modified to bypass your security. Incorrect initialization can occur anywhere, but it is probably most prevalent in rarely-encountered conditions that cause your code to inadvertently skip initialization, such as obscure errors." Improper initialization is listed as one of the top twenty-five most dangerous programming errors by the SANS Institute. How would you find those? How much would your compiler help?

The SPARK Toolset identifies all variables that have not been initialized prior to being read. And if the SPARK automatic source code analysis finds no uninitialized variables, then there really are none!

In this Gem, we will first inject a couple of initialization errors into some of the Tokeneer code and then show how you can use the SPARK tools to find those errors.

Step-by-Step Instructions

Let us start by first running the Examiner, one of the SPARK tools, on some of the Tokeneer code in the correct version. Then we will inject an uninitialized variable error in the same code and see how the Examiner's data-flow analysis finds it for us.

Step 1: Run the Examiner on the correct version of the code

The code below is from the procedure GetStartAndEndTimeFromFile in auditlog.adb.

```
264   procedure GetStartAndEndTimeFromFile
265     (TheFile     : in out File.T;
266      Description :    out AuditTypes.DescriptionT)
267     --# global in out AuditSystemFault;
268     --# derives AuditSystemFault,
269     --#         TheFile          from *,
270     --#                               TheFile &
271     --#         Description      from TheFile;
272   is
273      OK : Boolean;
274      FirstTime : Clock.TimeTextT;
275      LastTime  : Clock.TimeTextT;
276      TimeCount : Natural; -- type Natural to match formal
277                           -- in call to GetString
278      TimeOK    : Boolean := True;
279
        ...
311   -------------------------------------------------------------
312   -- begin GetStartAndEndTimeFromFile
313   -------------------------------------------------------------
314   begin
315
316      FirstTime := Clock.PrintTime(Clock.ZeroTime);
317      LastTime := Clock.PrintTime(Clock.ZeroTime);
        ...
358      Description := ConvertTimesToText;
359
360   end GetStartAndEndTimeFromFile;
```

Run the Examiner on auditlog.adb and notice that it reports no errors.

Step 2: Inject an initialization error and find it using the Examiner

Let’s introduce two types of uninitialized variables into the code – variables that are never initialized and variables that may not be initialized.

Lines 316 and 317 initialize the variables FirstTime and LastTime. Put these two lines of code inside an if statement so that the variables are only initialized if the variable OK is true. The modified code is shown below.

```
316   if OK then
317       FirstTime := Clock.PrintTime(Clock.ZeroTime);
318       LastTime := Clock.PrintTime(Clock.ZeroTime);
319   end if;
```

The changes introduce a number of improper initialization errors:

```
   * The variable OK is not initialized before it is read;
   * The variables FirstTime and LastTime may not be initialized;
   * The uninitialized variables FirstTime, LastTime and OK may then
     influence the expression that assigns a value to the variable
     Description -- the variable OK is indirectly used in the
expression.
```

Rerun the Examiner and notice that the Examiner identifies all uninitialized variables as errors, and it differentiates between variables that are not initialized and variables that may not be initialized:

```
auditlog.adb:316:10:
   Flow Error  20 - Expression contains reference(s) to variable OK
which has an undefined value.
auditlog.adb:316:10:
   Flow Error  22 - Value of expression is invariant.
auditlog.adb:360:7:
   Flow Error 504 - Statement contains reference(s) to variable
FirstTime, which may have an
   undefined value.
auditlog.adb:360:7:
   Flow Error 504 - Statement contains reference(s) to variable
LastTime, which may have an
   undefined value.
auditlog.adb:362:8:
   Flow Error 602 - The undefined initial value of OK may be used in
the derivation
   of Description.
auditlog.adb:362:8:
   Flow Error 602 - The undefined initial value of FirstTime may be
used in the
  derivation of Description.
auditlog.adb:362:8:
   Flow Error 602 - The undefined initial value of LastTime may be used
in the
   derivation of Description.
```

Notice also that the compiler already warns against the simplest of these improper initialization errors:

```
auditlog.adb:316:04: warning: "OK" may be referenced before it has a
value
```

Step 3: Find an array index initialization error

Another common error is accessing an array element with an uninitialized variable as the index. In this example we will inject such an error and demonstrate that the Examiner also finds these errors.

The code below is from the procedure DeleteLogFile in auditlog.adb.

```
512   procedure DeleteLogFile ( Index : LogFileIndexT)
513   --# global in out AuditSystemFault;
514   --#        in out LogFiles;
515   --#        in out LogFilesStatus;
516   --#        in out LogFileEntries;
517   --# derives AuditSystemFault,
518   --#         LogFiles         from *,
519   --#                               LogFiles,
520   --#                               Index &
```

```
521   --#         LogFilesStatus,
522   --#         LogFileEntries   from *,
523   --#                                Index;
524   is
525      OK : Boolean;
526      TheFile : File.T;
527   begin
528
529      TheFile := LogFiles (Index);
         ...
543   end DeleteLogFile;
```

Line 529 of the code accesses the Indexth element of the array LogFiles.

Declare a new variable I of type LogFileIndexT and replace Index, on line 529, with I. The modified code is shown below.

```
512     procedure DeleteLogFile ( Index : LogFileIndexT)
        ...
524     is
525         OK : Boolean;
526         TheFile : File.T;
527         I : LogFileIndexT;
528     begin
529
530         TheFile := LogFiles (I);
```

Run the Examiner on the file auditlog.adb and notice that it reports that the variable I has not been initialized:

```
auditlog.adb:530:18:
   Flow Error  20 - Expression contains reference(s) to variable I
which has an undefined value.
auditlog.adb:544:8:
   Flow Error  32 - The variable I is neither imported nor defined.
auditlog.adb:544:8:
   Flow Error  50 - AuditSystemFault is not derived from the imported
value(s) of Index.
auditlog.adb:544:8:
   Flow Error 602 - The undefined initial value of I may be used in the
derivation of AuditSystemFault.
auditlog.adb:544:8:
   Flow Error 602 - The undefined initial value of I may be used in the
derivation of LogFiles.
```

Again, in this very simple case, the compiler is able to spot the error too:

```
auditlog.adb:530:07: warning: variable "I" is read but never assigned
```

Although warnings from the compiler and warnings from the Examiner can sometimes overlap, one should keep in mind that the compiler warnings are heuristic, while the Examiner will warn against all possible improper initialization errors.

Summary

We have seen that the the SPARK tools can verify that a SPARK program is free from improper input validation errors. In the next Gem, we will study how SPARK handles a related class of errors — identifying ineffective statements.

## Related Source Code

**Gem #72: Tokeneer Discovery – Lesson 2**

**Author:** Dean Kuo and Angela Wallenburg, Praxis High Integrity Systems

**Abstract:** In the previous Gem in this series we saw how to deal with improper initialization in SPARK, based on source code from Tokeneer. In this Gem, we show how to identify ineffective statements.

**Let's get started…**

Every statement should have a purpose. An ineffective statement has no effect on any output variable and therefore has no effect on the behaviour of the code. The presence of ineffective statements reduces the quality and the maintainabiliy of the code. The SPARK Toolset identifies all ineffective statements.

In this Gem, we show how the SPARK Toolset finds ineffective statements to ensure that SPARK programs are free from them. We will inject an ineffective statement into the Tokeneer code and use the Examiner to locate it.

Step 1: Inject an ineffective statement

We will inject an ineffective statement into the implementation of the function NextListIndex in auditlog.adb.

```
       189  function NextListIndex(Value : LogFileIndexT) return
LogFileIndexT
       190  is
       191     Result : LogFileIndexT;
       192  begin
       193     if Value = LogFileIndexT'Last then
       194        Result := LogFileIndexT'First;
       195     else
       196        Result := Value + 1;
       197     end if;
       198     return Result;
       199  end NextListIndex;
```

Let's modify the above code by adding the new variable Result_Tmp of type LogFileTypeT (line 191) and change line 196 so that Value + 1 is assigned to the variable Result_Tmp instead of Result (see code below).

```
       189  function NextListIndex(Value : LogFileIndexT) return
LogFileIndexT
       190  is
       191     Result, Result_Tmp: LogFileIndexT;
       192  begin
       193     if Value = LogFileIndexT'Last then
       194        Result := LogFileIndexT'First;
       195     else
       196        Result_Tmp := Value + 1;
```

```
197     end if;
198     return Result;
199  end NextListIndex;
```

The statement on line 196 is ineffective because Result_Tmp is never used. Furthermore, the value for Result may be undefined when Value /= LogFileIndexT'Last.

Step 2: See how the Examiner finds the problem

Run the Examiner for auditlog.adb and, as expected, it finds the ineffective statement as well as the possible undefined value for Result.

```
auditlog.adb:196:10:
  Flow Error  10 - Ineffective statement.
auditlog.adb:198:14:
  Flow Error 501 - Expression contains reference(s) to variable Result,
which may have an undefined value.
auditlog.adb:199:8:
  Flow Error  33 - The variable Result_Tmp is neither referenced nor
exported.
auditlog.adb:199:8:
  Flow Error 602 - The undefined initial value of Result may be used in
the derivation of the function value.
```

Summary

In this Gem we have seen an example of the SPARK tools finding an ineffective statement. The SPARK tools will find all ineffective statements. In the next Gem we will study Input Validation.

**Related Source Code**

**Gem #73: Tokeneer Discovery – Lesson 3**

**Author:** Dean Kuo and Angela Wallenburg, Praxis High Integrity Systems

**Abstract:** In the previous Gem in this series, we saw how to identify ineffective statements in SPARK, based on source code from Tokeneer. In this Gem, we show how to validate input.

**Let's get started…**

Input validation ensures that your program's input conforms to expectations – for example, to ensure that the input has the right type. But validation requirements can be much more complicated than that. Incorrect input validation can lead to security and safety problems since many applications live in a "hostile" environment and the input might be constructed by an attacker. "It's the number one killer of healthy software…" according to the CWE/SANS list of the top twenty-five most dangerous programming errors.

For example, consider the following few lines of code from the original release of the Tokeneer code:

```
233      if Success and then
234         (RawDuration * 10 <= Integer(DurationT'Last) and
235          RawDuration * 10 >= Integer(DurationT'First))
then
236         Value := DurationT(RawDuration * 10);
237      else
```

This code has a check that the input RawDuration is in the right range before the value is updated – an example of so called defensive coding, according to the advice from the software experts who compiled the list of dangerous programming errors. Can you see the problem with this code?

The SPARK tools will identify a serious defect in this code, which could impact on security.

In this Gem, the above code will be investigated using the SPARK tools. This Gem shows the challenges in ensuring the absence of input validation errors, and the benefits of using SPARK to do so.

**Step-by-Step Instructions**

First, we will familiarize ourselves with two more advanced SPARK tools by running them on the correct version of the code. Then we inject the defect shown above into the Tokeneer code, rerun the SPARK tools, and interpret the results.

Step 1: Analyse the Correct Version of the Code

The correct lines of code are the following:

```
         233      if Success and then
         234         (RawDuration <= Integer(DurationT'Last) / 10 and
         235          RawDuration >= Integer(DurationT'First) / 10)
then
         236         Value := DurationT(RawDuration * 10);
         237      else
```

Analyse Tokeneer using the following steps:

* Run the Examiner on the file configdata.adb. * Run the Simplifier, a more advanced tool in the SPARK tool suite, which tries to show the absence of certain run-time errors by theorem proving. * Run POGS, which gives a summary of the verification just performed.

Now let us inspect the verification summary, where an overall summary of the Tokeneer verification is given. It shows that there are no errors which is expected as we ran the tools on the correct version of the code. Furthermore, it shows a number of tables describing details for the verification. For example, lines 2750 to 2766 of core.sum, shown below, are the results from the analysis of the procedure ReadDuration.

```
2750    VCs for procedure_readduration :
2751    ------------------------------------------------------------------------
2752         |       |                    |  -----Proved In-----  |       |       |
2753     #   | From  | To                 | vcg | siv | plg | prv | False | TO DO |
2754    ------------------------------------------------------------------------
2755     1   | start | rtc check @ 220    |     | YES |     |     |       |       |
2756     2   | start | rtc check @ 221    |     | YES |     |     |       |       |
2757     3   | start | rtc check @ 221    |     | YES |     |     |       |       |
2758     4   | start | rtc check @ 233    |     | YES |     |     |       |       |
2759     5   | start | rtc check @ 237    |     | YES |     |     |       |       |
2760     6   | start | rtc check @ 243    |     | YES |     |     |       |       |
2761     7   | start | rtc check @ 243    |     | YES |     |     |       |       |
2762     8   | start |    assert @ finish | YES |     |     |     |       |       |
2763     9   | start |    assert @ finish | YES |     |     |     |       |       |
2764     10  | start |    assert @ finish | YES |     |     |     |       |       |
2765     11  | start |    assert @ finish | YES |     |     |     |       |       |
2766    ------------------------------------------------------------------------
```

Note that the columns False and TO DO are empty, which means that the SPARK tools found no errors in the verification of procedure ReadDuration.

Step 2: Inject Erroneous Input Validation Code

Now, replace lines 234 and 235 in the file configdata.adb with the erroneous code:

```
         233      if Success and then
         234         (RawDuration * 10 <= Integer(DurationT'Last) and
         235          RawDuration * 10 >= Integer(DurationT'First))
then
         236         Value := DurationT(RawDuration * 10);
         237      else
```

Essentially this code concerns the validation of an input – an integer value RawDuration – that is read from a file, and is expected to be in the range 0..200 seconds before it is converted into a number of tenths of seconds in the range 0..2000.

Step 3: Re-Analyse the Faulty Code

Re-analyse Tokeneer.

The results from the analysis of the procedure ReadDuration is shown below.

```
2750    VCs for procedure_readduration :
2751    ------------------------------------------------------------------------------
2752          |       |                     |  -----Proved In-----  |       |       |
2753     #    | From  | To                  | vcg | siv | plg | prv | False | TO DO |
2754    ------------------------------------------------------------------------------
2755     1    | start | rtc check @ 220     |     | YES |     |     |       |       |
2756     2    | start | rtc check @ 221     |     | YES |     |     |       |       |
2757     3    | start | rtc check @ 221     |     | YES |     |     |       |       |
2758     4    | start | rtc check @ 233     |     |     |     |     |       |  YES  |
2759     5    | start | rtc check @ 237     |     | YES |     |     |       |       |
2760     6    | start | rtc check @ 243     |     | YES |     |     |       |       |
2761     7    | start | rtc check @ 243     |     | YES |     |     |       |       |
2762     8    | start |    assert @ finish  | YES |     |     |     |       |       |
2763     9    | start |    assert @ finish  | YES |     |     |     |       |       |
2764     10   | start |    assert @ finish  | YES |     |     |     |       |       |
2765     11   | start |    assert @ finish  | YES |     |     |     |       |       |
2766    ------------------------------------------------------------------------------
```

Notice that this time there is one YES in the TO DO column. The SPARK Toolset has detected a potential problem with the procedure ReadDuration.

Step 4: Investigate the Verification Output

Now, let us have look into what the error that the SPARK tools have found really means.

The file readduration.siv contains the Simplifier's analysis of the procedure. Lines 38 to 55 (shown below) of the file show the potential problem the SPARK Toolset has identified. The Simplifier, on line 54, is trying to check no arithmetic overflow errors will occur when evaluating the expression RawDuration * 10 – that is, when Success is True then RawDuration * 10 >= -2147483648 (Integer'First) and RawDuration * 10 <= 2147483648 (Integer'Last).

```
38      procedure_readduration_4.
39      H1:    rawduration__1 >= - 2147483648 .
40      H2:    rawduration__1 <= 2147483647 .
    ...
52              ->
53      C1:    success__1 -> rawduration__1 * 10 >= - 2147483648 and
rawduration__1 *
54                 10 <= 2147483647 .
55
```

Here the SPARK theorem prover is trying to prove that RawDuration times 10 is within the limits of Integer, assuming only that it was within the limits before it was multiplied

by 10. This should not be possible to prove. Think about a scenario where Tokeneer was given an input floppy disk where RawDuration was set to 1000000000. Both the assumptions H1 and H2 would be true, but C1 – the conclusion – would be false!

This is a serious defect since a malicious user holding the “security officer” role can deliberately attack the system by supplying a file that contains a malformed configuration data file – one that contains a value for RawDuration that is greater than Integer’Last/10.

Summary

In SPARK, developers need to be explicit about the intended input and output of program components. This has the benefit of the SPARK tools being able to automatically find defects that are hard to prevent, hard to detect, and with important security consequences. In this Gem we have seen this exemplified with input validation. In the next Gem, we will look into SPARK contracts.

## Related Source Code

**Gem#74: Tokeneer Discovery – Lesson 4**

**Author:** Dean Kuo and Angela Wallenburg, Praxis High Integrity Systems

**Abstract:** In the previous Gem in this series, we saw how to validate input in SPARK, based on source code from Tokeneer. In this Gem, we show how the SPARK Toolset can verify application-specific safety and security properties.

**Let's get started…**

In this Gem, we will see how the SPARK tools detect any differences between a program's intended behaviour, as specified in its contract, and its actual behaviour, as implemented in the code. Thus the SPARK tools can be used either to find defects in the contract, to find defects in the implementation, to find defects in both, or to show conformance between intended and actual behaviour.

**Step-by-Step Instructions**

Step 1: Analyse the Correct Version of the Code

The precondition for procedure AddElementToCurrentFile in auditlog.adb (lines 772 – 774 in the code below) specifies that the array element LogFileEntries (CurrentLogFile) is less than MaxLogFileEntries and the postcondition specifies that the array element is incremented by 1.

```
751         procedure AddElementToCurrentFile
752           (ElementID    : in     AuditTypes.ElementT;
753            Severity     : in     AuditTypes.SeverityT;
754            User         : in     AuditTypes.UserTextT;
755            Description  : in     AuditTypes.DescriptionT)
756           --# global in     Clock.Now;
757           --#        in     CurrentLogFile;
758           --#        in out AuditSystemFault;
759           --#        in out LogFiles;
760           --#        in out LogFileEntries;
761           --# derives AuditSystemFault,
762           --#         LogFiles         from *,
763           --#                                Description,
764           --#                                LogFiles,
765           --#                                Clock.Now,
766           --#                                ElementID,
767           --#                                Severity,
768           --#                                User,
769           --#                                CurrentLogFile &
770           --#         LogFileEntries   from *,
771           --#                                CurrentLogFile;
772           --# pre LogFileEntries(CurrentLogFile) <
MaxLogFileEntries;
773           --# post LogFileEntries(CurrentLogFile) =
774           --#             LogFileEntries~(CurrentLogFile) + 1;
775
```

```
776         is
777            TheFile : File.T ;
778         begin
779            TheFile := LogFiles (CurrentLogFile);
780            AddElementToFile
781              (TheFile => TheFile,
782               ElementID    => ElementID,
783               Severity     => Severity,
784               User         => User,
785               Description  => Description);
786            LogFiles (CurrentLogFile) := TheFile;
787
788            LogFileEntries(CurrentLogFile) :=
LogFileEntries(CurrentLogFile) + 1;
789         end AddElementToCurrentFile;
```

Analyse Tokeneer with the SPARK Toolset. The result of the analysis shows (see below) that the SPARK Toolset has identified no problems with the code in the procedure AddElementToLogFile – the columns False and TO DO are empty.

```
659    VCs for procedure_addelementtocurrentfile :
660    ------------------------------------------------------------------------------
661          |      |                    |  -----Proved In-----  |       |       |
662     #    | From  | To                | vcg | siv | plg | prv | False | TO DO |
663    ------------------------------------------------------------------------------
664     1    | start | rtc check @ 781    |     | YES |     |     |       |       |
665     2    | start | rtc check @ 782    |     | YES |     |     |       |       |
666     3    | start | rtc check @ 788    |     | YES |     |     |       |       |
667     4    | start | rtc check @ 790    |     | YES |     |     |       |       |
668     5    | start |    assert @ finish  |     | YES |     |     |       |       |
669    ------------------------------------------------------------------------------
```

Step 2: Change the Contract – But not the Implementation

Let us change the postcondition so the procedure's contract no longer matches its implementation. The SPARK Toolset will then detect the inconsistency. Change the postcondition to specify that the array element LogFileEntries(CurrentLogFile) should be incremented by 10. The modified code is shown below.

```
772          --# pre LogFileEntries(CurrentLogFile) <
MaxLogFileEntries;
773          --# post LogFileEntries(CurrentLogFile) =
774          --#             LogFileEntries~(CurrentLogFile) +
10;
775
```

Step 3: Re-Analyse and Study the Results

Re-analyse Tokeneer. The results of the analysis for the procedure AddElementToLogFile has changed – the columns False and TO DO are no longer empty (see below).

```
659    VCs for procedure_addelementtocurrentfile :
660    ------------------------------------------------------------------------------
661          |      |                    |  -----Proved In-----  |       |       |
```

```
662    #    | From  | To                 | vcg | siv | plg | prv | False | TO DO |
663    ------------------------------------------------------------------------------
664     1    | start | rtc check @ 781     |     | YES |     |     |       |       |
665     2    | start | rtc check @ 782     |     | YES |     |     |       |       |
666     3    | start | rtc check @ 788     |     | YES |     |     |       |       |
667     4    | start | rtc check @ 790     |     | YES |     |     |       |       |
668     5    | start |    assert @ finish  |     |     |     |     |  YES  |       |
669    ------------------------------------------------------------------------------
```

The SPARK Toolset has identified a potential problem with the code. The problem is that the procedure's contract and implementation don't match.

Step 4: Revert the Contract – But Change the Implementation

Undo the changes to the postcondition and change line 788 so that the input value is preserved.

```
     774           --# pre LogFileEntries(CurrentLogFile) <
MaxLogFileEntries;
     775           --# post LogFileEntries(CurrentLogFile) =
     776           --#             LogFileEntries~(CurrentLogFile) + 1
                     ...
     791           LogFileEntries(CurrentLogFile) :=
LogFileEntries(CurrentLogFile) + 0;
     792        end AddElementToCurrentFile;
```

Re-analyse Tokeneer. The analysis of the procedure is unchanged, as the implementation, like previously, does not match the procedure's contract.

Step 5: Strengthen the Contract

Revert the code to its original state.

In SPARK, we can strengthen the procedure's contract and say more about the properties of the procedure. Let's add extra code assigning the value 10 to the first element of the array LogFileEntries if CurrentLogFile is not LogFileIndexType'First (lines 789 – 791 below).

```
      788         LogFileEntries(CurrentLogFile) :=
LogFileEntries(CurrentLogFile) + 1;
      789         if CurrentLogFile /= LogFileIndexType'First then
      790            LogFileEntries(LogFileIndexType'First) := 10;
      791         end if;
      792      end AddElementToCurrentFile;
```

Re-analyse Tokeneer and notice that no errors are reported, as the procedure's implementation is not inconsistent with its contract. The postcondition says nothing about the effects of the procedure on any of the array elements except the one indexed by CurrentLogEntry.

We can strengthen the postcondition (lines 775 and 776 below) to specify that only the entry indexed by CurrentLogFile is incremented and all other elements remain unchanged.

```
 772   --# pre LogFileEntries(CurrentLogFile) < MaxLogFileEntries;
 773   --# post LogFileEntries = LogFileEntries~[CurrentLogFile =>
LogFileEntries~(CurrentLogFile)+1];
```

Re-analyse Tokeneer and notice, on lines 659 to 671, that the mismatch between the implementation and contract has been detected.

```
 657    VCs for procedure_addelementtocurrentfile :
 658    ------------------------------------------------------------------------------
 659          |       |                     |  -----Proved In-----  |       |       |
 660     #    | From  | To                  | vcg | siv | plg | prv | False | TO DO |
 661    ------------------------------------------------------------------------------
 662     1    | start | rtc check @ 783     |     | YES |     |     |       |       |
 663     2    | start | rtc check @ 784     |     | YES |     |     |       |       |
 664     3    | start | rtc check @ 790     |     | YES |     |     |       |       |
 665     4    | start | rtc check @ 792     |     | YES |     |     |       |       |
 666     5    | start | rtc check @ 794     |     | YES |     |     |       |       |
 667     6    | start |    assert @ finish  |     |     |     |     |       |  YES  |
 668     7    | start |    assert @ finish  |     | YES |     |     |       |       |
 669    ------------------------------------------------------------------------------
```

Summary

This Gem demonstrates that the more precise the specification, the more bugs the SPARK Toolset can detect. The use of the SPARK Toolset during development to verify code is, in our experience, more effective than compiling and testing, since the analysis is for all input data and not just a few specific test cases. In the next Gem we will see how SPARK deals with overflow errors.

## Related Source Code

**Gem #75: Tokeneer Discovery – Lesson 5**

**Author:** Dean Kuo and Angela Wallenburg, Praxis High Integrity Systems

**Abstract:** In the previous Gem in this series, we saw how the SPARK Toolset can verify application-specific safety and security properties, based on source code from Tokeneer. In this Gem, we show how to deal with overflow errors.

**Let's get started…**

An overflow error occurs when the capacity of a device is exceeded. Overflow errors are a source of quality and security concerns. For instance, when an arithmetic overflow occurs, a calculated value does not fit in its specified size, and the calculation (and the program) just stops. Buffer overflow happens when a process stores data in a buffer outside of the memory that the programmer set aside for it. Buffer overflow errors are widely known to present a vulnerability to malicious hackers, who might exploit the error to sneak their own code onto a victim's disk, storing it outside of the intended buffer.

The SPARK tools detect all potential arithmetic and buffer overflow errors. In the Gem about input validation, we saw an example of an arithmetic overflow. In this Gem, we will study how the SPARK tools find a buffer overflow error that we have injected into the Tokeneer code.

**Step-by-Step Instructions**

We will introduce a buffer overflow error into auditlog.adb and show how the SPARK Toolset detects it.

Step 1: Inject a Buffer Overflow Error

The procedure AddElementToCurrentFile increments the element indexed by CurrentLogFile in the array LogFileEntries – see below.

```
      775  --# post LogFileEntries(CurrentLogFile) =
      776  --#             LogFileEntries~(CurrentLogFile) + 1;
              ...
      790     LogFileEntries(CurrentLogFile) :=
LogFileEntries(CurrentLogFile) + 1;
```

Change the code on line 790 and the postcondition on 775 to 776 so the procedure copies the value in the CurrentLogFile+1th element into the CurrentLogFileth element of the array. The modified code is shown below.

```
      774  --# pre LogFileEntries(CurrentLogFile) < MaxLogFileEntries;
      775  --# post LogFileEntries(CurrentLogFile) =
      776  --#             LogFileEntries~(CurrentLogFile+1);
           ...
```

```
      790  LogFileEntries(CurrentLogFile) :=
LogFileEntries(CurrentLogFile+1) ;
```

Step 2: Analyse and Study the Verification Output

Analyse Tokeneer. The SPARK Toolset identifies, on lines 587 to 597 (see below), that there is a potential problem with the procedure AddElementToCurrentFile.

```
659    VCs for procedure_addelementtocurrentfile :
660    ------------------------------------------------------------------------------
661          |      |                     |  -----Proved In-----  |       |       |
662    #     | From  | To                 | vcg | siv | plg | prv | False | TO DO |
663    ------------------------------------------------------------------------------
664     1    | start | rtc check @ 781    |     | YES |     |     |       |       |
665     2    | start | rtc check @ 782    |     | YES |     |     |       |       |
666     3    | start | rtc check @ 788    |     | YES |     |     |       |       |
667     4    | start | rtc check @ 790    |     |     |     |     |       |  YES  |
668     5    | start |    assert @ finish |     | YES |     |     |       |       |
669    ------------------------------------------------------------------------------
```

The Simplifier failed to show that CurrentLogFile+1 is always within range of the index type, because it is not true – it goes outside its range when CurrentLogFile = LogFileIndexType'Last, which then causes a buffer overflow.

Summary

Buffer overflow errors are common and present a security vulnerability. The SPARK Toolset can verify that a SPARK program is free from arithmetic as well as buffer overflow errors. In this Gem we have seen how the SPARK tools can be used to detect a buffer overflow error. In the next Gem, we will see how SPARK can be used in Ensuring Secure Information Flow.

## Related Source Code

**Gem #76: Tokeneer Discovery – Lesson 6**

**Author:** Dean Kuo and Angela Wallenburg, Praxis High Integrity Systems

**Abstract:** In the previous Gem in this series, we saw how to deal with overflow errors, based on source code from Tokeneer. In this Gem, we show how to ensure secure information flow.
This Gem brings us to the end of this series on Tokeneer, and we would like to say a big thank you to the SPARK team at Praxis HIS for providing these resources.

**Let's get started…**

Error message information leak occurs when secure data is leaked, through error messages, to unauthorised users, and is one of the top twenty-five most dangerous programming errors according to SANS Institute. The general problem is ensuring that information flow adheres to certain policies — for example, certain data should never be written in an error message to a log file that may be accessible by unauthorised users.

The objective of this Gem is to demonstrate that the Examiner detects information flow violations.

**Step-by-Step Instructions**

Step 1: Study a Contract from an Information Flow Perspective

The code below is from the procedure Verify in bio.adb. The out variable MatchResult returns the result of whether a person's fingerprint matched their template. The local variable NumericReturn is set to the enumerated value BioApiOk if the fingerprint successfully matched; otherwise it returns an error code.

When a match is unsuccessful, a log record is written including the variable NumericReturn, which is derived from the person's Template.

```
      221  procedure Verify(Template       : in
IandATypes.TemplateT;
      222                   MaxFAR         : in     IandATypes.FarT;
      223                   MatchResult    :    out
IandATypes.MatchResultT;
      224                   AchievedFAR    :    out IandATypes.FarT)
             ...
      230  --# derives AuditLog.State,
      231  --#         AuditLog.FileState from AuditLog.State,
      232  --#                                 AuditLog.FileState,
      233  --#                                 Template,
      234  --#                                 Clock.Now,
      235  --#                                 ConfigData.State,
      236  --#                                 Interface.Input &
             ...
      242  is
```

```
243     NumericReturn : BasicTypes.Unsigned32T;
244  begin
245     Interface.Verify(Template     => Template,
246                      MaxFAR       => MaxFAR,
247                      MatchResult  => MatchResult,
248                      AchievedFAR  => AchievedFAR,
249                      BioReturn    => NumericReturn);
250
251     if NumericReturn /= ValueOf(BioAPIOk) then
252        -- An error occurred, overwrite match information.

253        MatchResult := IandATypes.NoMatch;
254        AuditLog.AddElementToLog
255          (ElementID    => AuditTypes.SystemFault,
256           Severity     => AuditTypes.Warning,
257           User         => AuditTypes.NoUser,
258           Description  => MakeDescription ("Biometric device
failure ",
259                                           NumericReturn));
260        end if;
261  end Verify;
```

If the log were accessible to potential hackers, which is not the case for Tokeneer, then this would be an example of an error message information leak. Fingerprint templates should never be accessible by hackers.

Step 2: Change an Information Flow Aspect of the Contract

Change the contract for the procedure Verify to specify that no information written to the log is derived from Template by deleting line 233.

```
230   --# derives AuditLog.State,
231   --#         AuditLog.FileState from AuditLog.State,
232   --#                                 AuditLog.FileState,
233   --#
234   --#                                 Clock.Now,
235   --#                                 ConfigData.State,
236   --#                                 Interface.Input &
```

The corresponding line (line 73) of code needs to be removed from the file bio.ads.

```
60   procedure Verify(Template       : in
IandATypes.TemplateT;
61                    MaxFAR         : in     IandATypes.FarT;
62                    MatchResult    :    out
IandATypes.MatchResultT;
63                    AchievedFAR    :    out IandATypes.FarT);
     ...
69   --# derives AuditLog.State,
70   --#         AuditLog.FileState from Input,
71   --#                                 AuditLog.State,
72   --#                                 AuditLog.FileState,
73   --#
74   --#                                 Clock.Now,
```

```
      75   --#                                    ConfigData.State &
```

Step 3: Use the SPARK Tools to Detect the Information Leak

Examine the file bio.adb and notice the Examiner reports the error that information derived from the variable Template is written to the log. This means that data derived from the template is being written to the log!

```
bio.adb:261:8:
    Flow Error 601 - AuditLog.State may be derived from the imported
value(s) of Template.
bio.adb:261:8:
    Flow Error 601 - AuditLog.FileState may be derived from the
imported value(s) of Template.
```

Step 4: Introduce a Malicious Hack

The Examiner also detects when data has been incorrectly used. We now add back door code in the procedure Verify (lines 248-251 below). It returns a positive match independent of the user's fingerprint when the clock is at midnight.

```
      223  procedure Verify(Template       : in
IandATypes.TemplateT;
      224                   MaxFAR         : in     IandATypes.FarT;
      225                   MatchResult    :    out
IandATypes.MatchResultT;
      226                   AchievedFAR    :    out IandATypes.FarT)
            ...
      232  --# derives AuditLog.State,
      233  --#         AuditLog.FileState from AuditLog.State,
      234  --#                                 AuditLog.FileState,
      235  --#                                 Template,
      236  --#                                 Clock.Now,
      237  --#                                 ConfigData.State,
      238  --#                                 Interface.Input &
             ...
      244  is
      245     NumericReturn : BasicTypes.Unsigned32T;
      246     T             : Clock.TimeT;
      247  begin
      248     T := Clock.GetNow;
      249     if T = Clock.ZeroTime then
      250        MatchResult := IandATypes.Match;
      251        AchievedFAR := 0;
      252     else
      253
      254        Interface.Verify(Template     => Template,
      255                         MaxFAR       => MaxFAR,
      256                         MatchResult  => MatchResult,
      257                         AchievedFAR  => AchievedFAR,
      258                         BioReturn    => NumericReturn);
      259
            ...
      272    end if;
```

```
          ...
      275  end Verify;
```

Step 5: Use the SPARK Tools to Detect the Weakness

Examine the file bio_bad.adb and notice that the Examiner reports the error that MatchResult is dependent on the current time (Clock.now), which is inconsistent with the procedure's contract. The Examiner has identified an information flow inconsistency due to the presence of the back door.

```
bio_bad.adb:243:40:
    Flow Error   4 - The dependency of the exported value of
AuditLog.State on the imported value
    of  Verify.Template has not been previously stated.
bio_bad.adb:243:40:
    Flow Error   4 - The dependency of the exported value of
AuditLog.FileState on the imported value
    of Verify.Template has not been previously stated.
bio_bad.adb:275:8:
    Flow Error 601 - MatchResult may be derived from the imported
value(s) of Clock.Now.
bio_bad.adb:275:8:
    Flow Error 601 - AchievedFAR may be derived from the imported
value(s) of Clock.Now.
```

## Summary

In this Gem we have learnt about information flow contracts and we have seen the SPARK tools detect a malicious hack. SPARK programs are free from information leaks when the contract accurately specifies the desired information flow between variables.

## Related Source Code