SPARK

Quick Reference 1

Toolset and Annotations

Examiner

The key SPARK tool is the Examiner. It performs static semantic analysis, information-flow analysis and VC Generation.

Typical usages:

```
spark p.ads -l=p.lss p.adb -l=p.lsb
spark -w=all.wrn -i=project.idx @project.smf
```

Main options[EX_UM 3.1] (all before names of source or meta files):

`-i`	identify the index file to be used
`-w`	identify the warning file to be used
`-so`	gives the default source extension
`-l`	gives the listing file extension (nb -l after a file name specifies the listing file for that file only.)
`-rep`	provides the name of the report file
`-conf`	provides the name of the configuration file
`-plain`	plain output – suppresses dates, times, line numbers etc.
`-flow`	control type of flow analysis applied (-fl=i or -fl=d)
`-noe`	suppresses echo to standard output
`-vcg`	generate VCs
`-brief`	"brief" errors in same format as gcc compiler
`-rules`	controls generation of proof rules for composite constants
`-output`	specifies directory into which to create output files
`-noswitch`	ignore spark.sw file
`-debug`	enables various trace and debug options
`-help`	displays full list of all options and switches
`-version`	display Examiner version and exit

The @ symbol on the command line introduces a meta file.

Configuration Files [EX_UM 4.5]

A *configuration file* allows implementation-dependent values to be supplied to the Examiner. The syntax resembles a package declaration for packages Standard and/or System, although there are significant restrictions on the types of declaration allowed.

Example:

```
package Standard is
   type Integer is range –2**31 .. 2**31-1;
   type Short_Integer is range –32768 .. 32767;
   type Long_Integer is range –2**63 .. 2**63-1;
end Standard;
```

Simplifier

The main purpose of the Simplifier is to simplify verification conditions prior to developing a proof. In many cases the Simplifier is able to reduce all the conclusions to True.

sparksimp traverses the current working directory tree, finds all vcg or pfs files that need simplifying and applies the Simplifier to them.

Typical usage:

```
sparksimp
```

Main options [SIMP_UM 8]

`-a` - simplify all files regardless of timestamp
`-n` – find and report files but don't actually run the simplifier
`-l` - log screen output to *.log files
`-t` – sort VCG files, largest first
`-r` – reverse simplification order
`-p=N` – run N copies of the Simplifier in parallel

`-sargs` - pass all remaining arguments to each Simplifier run

spadesimp runs the Simplifier on the specified file.

Typical usage:

```
spadesimp mode
```

Main options [SIMP_UM 5]

`-plain` - suppress additional information in output

POGS (Proof Obligation Summary Tool)

This tool summarises the semantic output files produced by the Examiner, the Simplifier and the Proof Checker. It traverses the current working directory tree, placing its output file in the current directory.

Typical usage:

```
pogs
```

Main options [POGS_UM]

`-i` – ignore timestamps on siv files
`-p` – plain output – suppresses dates, times etc. in output
`-s` – short summary – suppresses per subprogram analysis in output
`-x` – XML output – note that this cannot be combined with /p or /s.

Annotations

Subprogram Annotations

The *global annotation* makes visible any global variables accessed by a subprogram [SPARK 6.1]:

Syntax:

```
--# global {[mode] variableName {, variableName};}
```

Examples:

```
procedure Push(Value: in Integer);
--# global in out Stack;

procedure Control;
--# global in Sensor.State;
--#        out Valve.State;
```

The *derives annotation* specifies the information flow between the parameters (and global variables) of a procedure [SPARK 6.1]:

Syntax:

```
--# derives [dependencyClause{&dependencyClause}];
```

Examples:

```
procedure Flt_Integrate(Fault   : in Boolean;
                        Trip    : in out Boolean;
                        Counter : in out Integer)
--# derives Trip    from *, Fault, Counter &
--#         Counter from *, Fault;
…

procedure BusyWait
--# derives ;
```

The *null derives* can be used when a subprogram *imports* variables but no visible *export* is derived from them [SPARK 6.1.2]:

Example:

```
--# derives null from X, Y, Z;
```

Package Annotations

The *own annotation* announces the constituents of the package state to other packages in the system [SPARK 7]:

Syntax:

```
--# own mode ownVariable {, mode ownVariable};
```

Example:

```
--# own X, Y, State;
```

Where a type declaration appears later in a package, a *type announcement* can be made.

Example:

```
--# own X: X_Type;
```

The *initialization specification* announces which unmoded *own variables* will be initialized at package elaboration - either at declaration or in the executable statements of the package body [SPARK 7]:

Syntax:
```
--# initializes ownVariable {, ownVariable};
```

Example:
```
package Random_Numbers
--# own Seed;
--# initializes Seed;
…
```

A proof *type annotation* can be supplied for *abstract own variables*:

Syntax [VCG 3.2.2]:
```
--# type identifier is abstract;
```

Example:
```
package The_Stack
--# own State : StackType;
--# initializes State;
is
--# type StackType is abstract;
…
```

A *refinement definition* appears in the package body - it expresses every abstract *own variable* in terms of its constituents [SPARK 7.2]:

Syntax:
```
--# own subject is constituent {, constituent};
```

Example:
```
package body The_Stack
--# own State is S, Pointer;
is
…
```

The *inherit clause* controls access to global entities outside of a package or the main subprogram[SPARK 7].
Syntax:
```
--# inherit packageName {, packageName};
```

Example:
```
package Q is
  …
end Q;

--# inherit Q;
package P is
  …
end P;
```

Library Units and the Main Program

SPARK compilation units consist of library units and subunits. A program consists of packages and a *single* library level subprogram. The *main program annotation* is given to this specific subprogram [SPARK 10.1]:

Example:
```
with S1, S2, S3;
--# inherit S1, S2, S3;
--# main_program;
procedure Main
is
begin
   …
end Main;
```

To ignore parts of a program during examination, the *hidden text annotation* can be used. This can serve a number of purposes - it enables parts of a program to be examined even if the whole cannot be compiled and it permits sections of the program (perhaps written in full Ada) to be excluded from analysis. Entities that may be hidden: subprogram body, package body, package specification private part, package body sequence of statements [SPARK M.1]:

Syntax:
```
--# hide identifier;
```

Example:
```
procedure Secret
--# global out This;
--#        in That;
--# derives This from That;
is
  --# hide Secret;
  …
end Secret;
```

A *type assertion* allows the user to specify the base type which the compiler will associate with a signed integer type. The base type must be supplied to the Examiner in the *configuration file*. This assertion allows the Examiner to generate VCs which are much more readily discharged [RTC 5.2].

Example:
```
type T is range 1 .. 10
--# assert T'Base is Short_Short_Integer;
```

References

[EX_UM] Examiner User Manual
[Simp_UM] SPADE Simplifier User Manual
[VCG] Generation of VCs for SPARK Programs
[RTC] Generation of RTCs for SPARK Programs
[SPARK] SPARK95 – The SPADE Ada95 Kernel

Verification Annotations

In order to prove the correctness of a program, certain hypotheses may be stated which assert conditions which must always be satisfied. These are given as further *pre- and post- condition annotations* in the subprogram specification. (To make use of these annotations one of the VC generation options must be used on the Examiner command line.) [VCG 3]

Syntax:
```
--# pre predicate;
--# post predicate;
```

Examples:
```
procedure Inc(X: in out T)
--# derives X from X;
--# pre X < T'Last;
…
procedure Exchange(X, Y: in out Float)
--# derives X from Y &
--#         Y from X;
--# post X = Y~ and Y = X~;
…
```

To provide the equivalent of a post-condition for a *function*, the *return annotation* can be used [VCG 3]:

Syntax:
```
--# return expression;
--# return identifier => predicate;
```

Examples:
```
function Inc(X: Integer) return Integer;
--# return X + 1;

function Max(X, Y: Integer) return Integer;
--# return M => (A >= B -> M = X) and
--#              (B >= A -> M = Y);
```

The *assert annotation* can be used to specify conditions that are to be true - of particular use when verifying programs containing loops. The assert annotation forms a *cutpoint* in the program flow-graph. [VCG 3]:

Syntax:
```
--# assert predicate;
```

Examples:
```
procedure Div(M, N: in Integer; Q, R: out Integer)
--# derives Q, R from M, N;
--# pre (M >= 0) and (N > 0);
--# post (M = Q * N + R) and (R < N) and (R >= 0);
is
begin
  Q := 0;  R := M;
  loop
    --# assert (M = Q * N + R) and (R >= 0);
    …
```