The HOL-SPARK Program Verification Environment

Stefan Berghofer
secunet Security Networks AG

October 24, 2026

Contents

1 Introduction **2**
- 1.1 SPARK . . . 2
- 1.2 HOL-SPARK . . . 3

2 Verifying an Example Program **6**
- 2.1 Importing SPARK VCs into Isabelle . . . 6
- 2.2 Proving the VCs . . . 10
- 2.3 Optimizing the proof . . . 11

3 Principles of VC generation **14**

4 HOL-SPARK Reference **19**
- 4.1 Commands . . . 19
- 4.2 Types . . . 21
  - 4.2.1 Integers . . . 21
  - 4.2.2 Enumeration types . . . 22
  - 4.2.3 Records . . . 24
  - 4.2.4 Arrays . . . 24
- 4.3 User-defined proof functions and types . . . 25

Chapter 1

Introduction

This document describes a link between Isabelle/HOL and the SPARK/Ada tool suite for the verification of high-integrity software. Using this link, verification problems can be tackled that are beyond reach of the proof tools currently available for SPARK. A number of examples can be found in the directory `HOL/SPARK/Examples` in the Isabelle distribution. An open-source version of the SPARK tool suite is available free of charge from libre.adacore.com.

In the remainder of §1, we give an introduction to SPARK and the HOL-SPARK link. The verification of an example program is described in §2. In §3, we explain the principles underlying the generation of verification conditions for SPARK programs. Finally, §4 describes the commands provided by the HOL-SPARK link, as well as the encoding of SPARK types in HOL.

1.1 SPARK

SPARK [1] is a subset of the Ada language that has been designed to allow verification of high-integrity software. It is missing certain features of Ada that can make programs difficult to verify, such as *access types*, *dynamic data structures*, and *recursion*. SPARK allows to prove absence of *run-time exceptions*, as well as *partial correctness* using pre- and postconditions. Loops can be annotated with *invariants*, and each procedure must have a *dataflow annotation*, specifying the dependencies of the output parameters on the input parameters of the procedure. Since SPARK annotations are just written as comments, SPARK programs can be compiled by an ordinary Ada compiler such as GNAT. SPARK comes with a number of tools, notably the *Examiner* that, given a SPARK program as an input, performs a *dataflow*

analysis and generates *verification conditions* (VCs) that must be proved in order for the program to be exception-free and partially correct. The VCs generated by the Examiner are formulae expressed in a language called FDL, which is first-order logic extended with arithmetic operators, arrays, records, and enumeration types. For example, the FDL expression

```
for_all(i: integer, ((i >= min) and (i <= max)) ->
  (element(a, [i]) = 0))
```

states that all elements of the array `a` with indices greater or equal to `min` and smaller or equal to `max` are 0. VCs are processed by another SPARK tool called the *Simplifier* that either completely solves VCs or transforms them into simpler, equivalent conditions. The latter VCs can then be processed using another tool called the *Proof Checker*. While the Simplifier tries to prove VCs in a completely automatic way, the Proof Checker requires user interaction, which enables it to prove formulae that are beyond the scope of the Simplifier. The steps that are required to manually prove a VC are recorded in a log file by the Proof Checker. Finally, this log file, together with the output of the other SPARK tools mentioned above, is read by a tool called POGS (**P**roof **O**bli**G**ation **S**ummariser) that produces a table mentioning for each VC the method by which it has been proved. In order to overcome the limitations of FDL and to express complex specifications, SPARK allows the user to declare so-called *proof functions*. The desired properties of such functions are described by postulating a set of rules that can be used by the Simplifier and Proof Checker [1, §11.7]. An obvious drawback of this approach is that incorrect rules can easily introduce inconsistencies.

1.2 HOL-SPARK

The HOL-SPARK verification environment, which is built on top of Isabelle's object logic HOL, is intended as an alternative to the SPARK Proof Checker, and improves on it in a number of ways. HOL-SPARK allows Isabelle to directly parse files generated by the Examiner and Simplifier, and provides a special proof command to conduct proofs of VCs, which can make use of the full power of Isabelle's rich collection of proof methods. Proofs can be conducted using Isabelle's graphical user interface, which makes it easy to navigate through larger proof scripts. Moreover, proof functions can be introduced in a *definitional* way, for example by using Isabelle's package for recursive functions, rather than by just stating their properties as axioms, which avoids introducing inconsistencies.

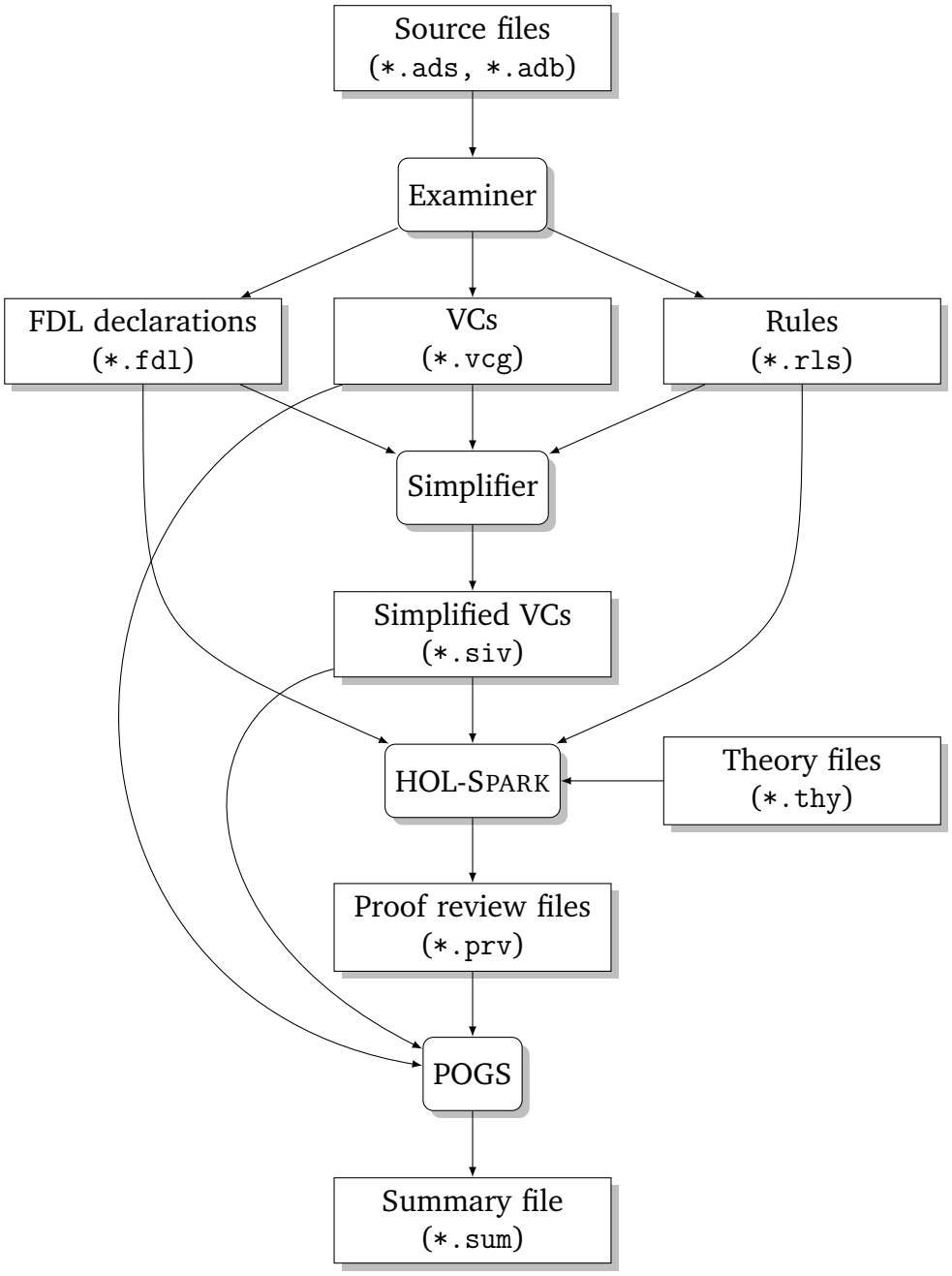


Figure 1.1: SPARK program verification tool chain

Figure 1.1 shows the integration of HOL-SPARK into the tool chain for the verification of SPARK programs. HOL-SPARK processes declarations (`*.fdl`) and rules (`*.rls`) produced by the Examiner, as well as simplified VCs (`*.siv`) produced by the SPARK Simplifier. Alternatively, the original unsimplified VCs (`*.vcg`) produced by the Examiner can be used as well. Processing of the SPARK files is triggered by an Isabelle theory file (`*.thy`), which also contains the proofs for the VCs contained in the `*.siv` or `*.vcg` files. Once that all verification conditions have been successfully proved, Isabelle generates a proof review file (`*.prv`) notifying the POGS tool of the VCs that have been discharged.

Chapter 2

Verifying an Example Program

We will now explain the usage of the SPARK verification environment by proving the correctness of an example program. As an example, we use a program for computing the *greatest common divisor* of two natural numbers shown in Fig. 2.1, which has been taken from the book about SPARK by Barnes [1, §11.6].

2.1 Importing SPARK VCs into Isabelle

In order to specify that the SPARK procedure `G_C_D` behaves like its mathematical counterpart, Barnes introduces a *proof function* `Gcd` in the package specification. Invoking the SPARK Examiner and Simplifier on this program yields a file `g_c_d.siv` containing the simplified VCs, as well as files `g_c_d.fdl` and `g_c_d.rls`, containing FDL declarations and rules, respectively. The files generated by SPARK are assumed to reside in the subdirectory `greatest_common_divisor`. For `G_C_D` the Examiner generates ten VCs, eight of which are proved automatically by the Simplifier. We now show how to prove the remaining two VCs interactively using HOL-SPARK. For this purpose, we create a *theory* `Greatest_Common_Divisor`, which is shown in Fig. 2.2. A theory file always starts with the keyword **theory** followed by the name of the theory, which must be the same as the file name. The theory name is followed by the keyword **imports** and a list of theories imported by the current theory. All theories using the HOL-SPARK verification environment must import the theory `SPARK`. In addition, we also include the `GCD` theory. The list of imported theories is followed by the **begin** keyword. In order to interactively process the theory shown in Fig. 2.2, we start Isabelle with the command

```
isabelle jedit -l HOL-SPARK Greatest_Common_Divisor.thy
```

```
package Greatest_Common_Divisor
is

   --# function Gcd(A, B: Natural) return Natural;

   procedure G_C_D(M, N: in Natural; G: out Natural);
     --# derives G from M, N;
     --# pre M >= 0 and N > 0;
     --# post G = Gcd(M,N);

end Greatest_Common_Divisor;

package body Greatest_Common_Divisor
is

   procedure G_C_D(M, N: in Natural; G: out Natural)
   is
      C, D, R: Integer;
   begin
      C := M; D := N;
      while D /= 0 loop
         --# assert C >= 0 and D > 0 and Gcd(C, D) = Gcd(M, N);
         R := C rem D;
         C := D; D := R;
      end loop;
      G := C;
   end G_C_D;

end Greatest_Common_Divisor;
```

Figure 2.1: SPARK program for computing the greatest common divisor

```
theory Greatest_Common_Divisor
imports "HOL-SPARK.SPARK"
begin

spark_proof_functions
  gcd = "gcd :: int ⇒ int ⇒ int"

spark_open ‹greatest_common_divisor/g_c_d›

spark_vc procedure_g_c_d_4
⟨proof⟩

spark_vc procedure_g_c_d_11
⟨proof⟩

spark_end

end
```

Figure 2.2: Correctness proof for the greatest common divisor program

The option "`-l HOL-SPARK`" instructs Isabelle to load the right object logic image containing the verification environment. Each proof function occurring in the specification of a SPARK program must be linked with a corresponding Isabelle function. This is accomplished by the command **spark_proof_functions**, which expects a list of equations of the form *name* = *term*, where *name* is the name of the proof function and *term* is the corresponding Isabelle term. In the case of `gcd`, both the SPARK proof function and its Isabelle counterpart happen to have the same name. Isabelle checks that the type of the term linked with a proof function agrees with the type of the function declared in the `*.fdl` file. It is worth noting that the **spark_proof_functions** command can be invoked both outside, i.e. before **spark_open**, and inside the environment, i.e. after **spark_open**, but before any **spark_vc** command. The former variant is useful when having to declare proof functions that are shared by several procedures, whereas the latter has the advantage that the type of the proof function can be checked immediately, since the VCs, and hence also the declarations of proof functions in the `*.fdl` file have already been loaded. We now instruct Isabelle to open a new verification environment and load a set of VCs. This is done using the command **spark_open**, which must be given the name of a `*.siv` file as an argument. Behind the scenes, Isabelle parses this file and the cor-

```
Context:

 fixes    m :: "int"
 and      n :: "int"
 and      c :: "int"
 and      d :: "int"
 assumes  g_c_d_rules1: "0 ≤ integer__size"
 and      g_c_d_rules6: "0 ≤ natural__size"
 notes definition
   defns = ‘integer__first = - 2147483648‘
     ‘integer__last = 2147483647‘
     ...

Definitions:

 g_c_d_rules2:  integer__first = - 2147483648
 g_c_d_rules3:  integer__last = 2147483647
 ...

Verification conditions:

path(s) from assertion of line 10 to assertion of line 10

procedure_g_c_d_4 (unproved)
   assumes  H1: "0 ≤ c"
   and      H2: "0 < d"
   and      H3: "gcd c d = gcd m n"
   ...
   shows    "0 < c - c sdiv d * d"
   and      "gcd d (c - c sdiv d * d) = gcd m n

path(s) from assertion of line 10 to finish

procedure_g_c_d_11 (unproved)
   assumes  H1: "0 ≤ c"
   and      H2: "0 < d"
   and      H3: "gcd c d = gcd m n"
   ...
   shows    "d = gcd m n"
```

Figure 2.3: Output of **spark_status** for `g_c_d.siv`

responding `*.fdl` and `*.rls` files, and converts the VCs to Isabelle terms. Using the command **spark_status**, the user can display the current VCs together with their status (proved, unproved). The variants **spark_status** `(proved)` and **spark_status** `(unproved)` show only proved and unproved VCs, respectively. For `g_c_d.siv`, the output of **spark_status** is shown in Fig. 2.3. To minimize the number of assumptions, and hence the size of the VCs, FDL rules of the form "`... may_be_replaced_by ...`" are turned into native Isabelle definitions, whereas other rules are modelled as assumptions.

2.2 Proving the VCs

The two open VCs are `procedure_g_c_d_4` and `procedure_g_c_d_11`, both of which contain the `gcd` proof function that the SPARK Simplifier does not know anything about. The proof of a particular VC can be started with the **spark_vc** command, which is similar to the standard **lemma** and **theorem** commands, with the difference that it only takes a name of a VC but no formula as an argument. A VC can have several conclusions that can be referenced by the identifiers `?C1`, `?C2`, etc. If there is just one conclusion, it can also be referenced by `?thesis`. It is important to note that the `div` operator of FDL behaves differently from the `div` operator of Isabelle/HOL on negative numbers. The former always truncates towards zero, whereas the latter truncates towards minus infinity. This is why the FDL `div` operator is mapped to the `sdiv` operator in Isabelle/HOL, which is defined as

```
a sdiv b = sgn a * sgn b * (|a| div |b|)
```

For example, we have that `- 5 sdiv 4 = - 1`, but `- 5 div 4 = - 2`. For non-negative dividend and divisor, `sdiv` is equivalent to `div`, as witnessed by theorem `sdiv_pos_pos`:

```
0 ≤ a ⟹ 0 ≤ b ⟹ a sdiv b = a div b
```

In contrast, the behaviour of the FDL `mod` operator is equivalent to the one of Isabelle/HOL. Moreover, since FDL has no counterpart of the SPARK operator **rem**, the SPARK expression `c` **rem** `d` just becomes `c - c sdiv d * d` in Isabelle. The first conclusion of `procedure_g_c_d_4` requires us to prove that the remainder of `c` and `d` is greater than `0`. To do this, we use the theorem `minus_div_mult_eq_mod [symmetric]` describing the correspondence between `div` and `mod`

```
a mod b = a - a div b * b
```

together with the theorem `pos_mod_sign` saying that the result of the `mod` operator is non-negative when applied to a non-negative divisor:

```
0 < l ⟹ 0 ≤ k mod l
```

We will also need the aforementioned theorem `sdiv_pos_pos` in order for the standard Isabelle/HOL theorems about `div` to be applicable to the VC, which is formulated using `sdiv` rather that `div`. Note that the proof uses `‘0 ≤ c‘` and `‘0 < d‘` rather than `H1` and `H2` to refer to the hypotheses of the current VC. While the latter variant seems more compact, it is not particularly robust, since the numbering of hypotheses can easily change if the corresponding program is modified, making the proof script hard to adjust when there are many hypotheses. Moreover, proof scripts using abbreviations like `H1` and `H2` are hard to read without assistance from Isabelle. The second conclusion of `procedure_g_c_d_4` requires us to prove that the `gcd` of `d` and the remainder of `c` and `d` is equal to the `gcd` of the original input values `m` and `n`, which is the actual *invariant* of the procedure. This is a consequence of theorem `gcd_non_0_int`

```
0 < y ⟹ gcd x y = gcd y (x mod y)
```

Again, we also need theorems `minus_div_mult_eq_mod [symmetric]` and `sdiv_pos_pos` to justify that SPARK's **rem** operator is equivalent to Isabelle's `mod` operator for non-negative operands. The VC `procedure_g_c_d_11` says that if the loop invariant holds before the last iteration of the loop, the post-condition of the procedure will hold after execution of the loop body. To prove this, we observe that the remainder of `c` and `d`, and hence `c mod d` is `0` when exiting the loop. This implies that `gcd c d = d`, since `c` is divisible by `d`, so the conclusion follows using the assumption `gcd c d = gcd m n`. This concludes the proofs of the open VCs, and hence the SPARK verification environment can be closed using the command **spark_end**. This command checks that all VCs have been proved and issues an error message if there are remaining unproved VCs. Moreover, Isabelle checks that there is no open SPARK verification environment when the final **end** command of a theory is encountered.

2.3 Optimizing the proof

When looking at the program from Fig. 2.1 once again, several optimizations come to mind. First of all, like the input parameters of the procedure, the local variables `C`, `D`, and `R` can be declared as `Natural` rather than

```
package body Simple_Greatest_Common_Divisor
is

   procedure G_C_D (M, N : in Natural; G : out Natural)
   is
      C, D, R : Natural;
   begin
      C := M; D := N;
      while D /= 0
        --# assert Gcd (C, D) = Gcd (M, N);
      loop
         R := C mod D;
         C := D; D := R;
      end loop;
      G := C;
   end G_C_D;

end Simple_Greatest_Common_Divisor;
```

```
theory Simple_Greatest_Common_Divisor
imports "HOL-SPARK.SPARK"
begin

spark_proof_functions
  gcd = "gcd :: int ⇒ int ⇒ int"

spark_open ‹simple_greatest_common_divisor/g_c_d›

spark_vc procedure_g_c_d_4
  ⟨proof⟩

spark_vc procedure_g_c_d_9
  ⟨proof⟩

spark_end

end
```

Figure 2.4: Simplified greatest common divisor program and proof

`Integer`. Since natural numbers are non-negative by construction, the values computed by the algorithm are trivially proved to be non-negative. Since we are working with non-negative numbers, we can also just use SPARK's **mod** operator instead of **rem**, which spares us an application of theorems `minus_div_mult_eq_mod [symmetric]` and `sdiv_pos_pos`. Finally, as noted by Barnes [1, §11.5], we can simplify matters by placing the **assert** statement between **while** and **loop** rather than directly after the **loop**. In the former case, the loop invariant has to be proved only once, whereas in the latter case, it has to be proved twice: since the **assert** occurs after the check of the exit condition, the invariant has to be proved for the path from the **assert** statement to the **assert** statement, and for the path from the **assert** statement to the postcondition. In the case of the `G_C_D` procedure, this might not seem particularly problematic, since the proof of the invariant is very simple, but it can unnecessarily complicate matters if the proof of the invariant is non-trivial. The simplified program for computing the greatest common divisor, together with its correctness proof, is shown in Fig. 2.4. Since the package specification has not changed, we only show the body of the packages. The two VCs can now be proved by a single application of Isabelle's proof method `simp`.

Chapter 3

Principles of VC generation

In this section, we will discuss some aspects of VC generation that are useful for understanding and optimizing the VCs produced by the SPARK Examiner.

As explained by Barnes [1, §11.5], the SPARK Examiner unfolds the loop

```
for I in T range L .. U loop
  --# assert P (I);
  S;
end loop;
```

to

```
if L <= U then
  I := L;
  loop
    --# assert P (I);
    S;
    exit when I = U;
    I := I + 1;
  end loop;
end if;
```

Due to this treatment of for-loops, the user essentially has to prove twice that S preserves the invariant *P*, namely for the path from the assertion to the assertion and from the assertion to the next cut point following the loop. The preservation of the invariant has to be proved even more often when the loop is followed by an if-statement. For trivial invariants, this might not seem like a big problem, but in realistic applications, where invariants are complex, this can be a major inconvenience. Often, the proofs of the invariant differ only in a few variables, so it is tempting to just copy and modify existing proof scripts, but this leads to proofs that are hard to

```
package Loop_Invariant
is

   type Word32 is mod 2 ** 32;

   procedure Proc1 (A : in Natural; B : in Word32; C : out Word32);
   --# derives C from A, B;
   --# post Word32 (A) * B = C;

   procedure Proc2 (A : in Natural; B : in Word32; C : out Word32);
   --# derives C from A, B;
   --# post Word32 (A) * B = C;

end Loop_Invariant;

package body Loop_Invariant
is

   procedure Proc1 (A : in Natural; B : in Word32; C : out Word32)
   is
   begin
      C := 0;
      for I in Natural range 1 .. A
        --# assert Word32 (I - 1) * B = C;
      loop
         C := C + B;
      end loop;
   end Proc1;

   procedure Proc2 (A : in Natural; B : in Word32; C : out Word32)
   is
   begin
      C := 0;
      for I in Natural range 1 .. A
        --# assert Word32 (I - 1) * B = C;
      loop
         C := C + B;
        --# assert Word32 (I) * B = C;
      end loop;
   end Proc2;

end Loop_Invariant;
```

Figure 3.1: Assertions in for-loops

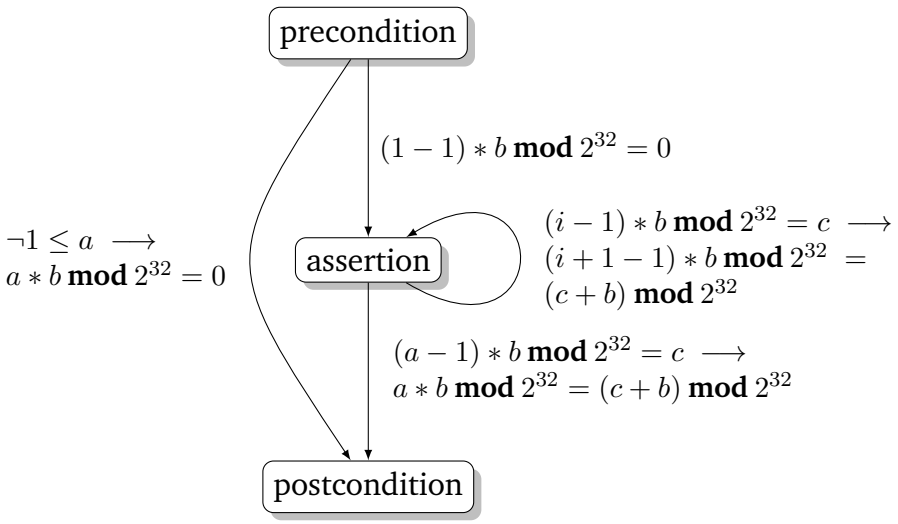


Figure 3.2: Control flow graph for procedure `Proc1`

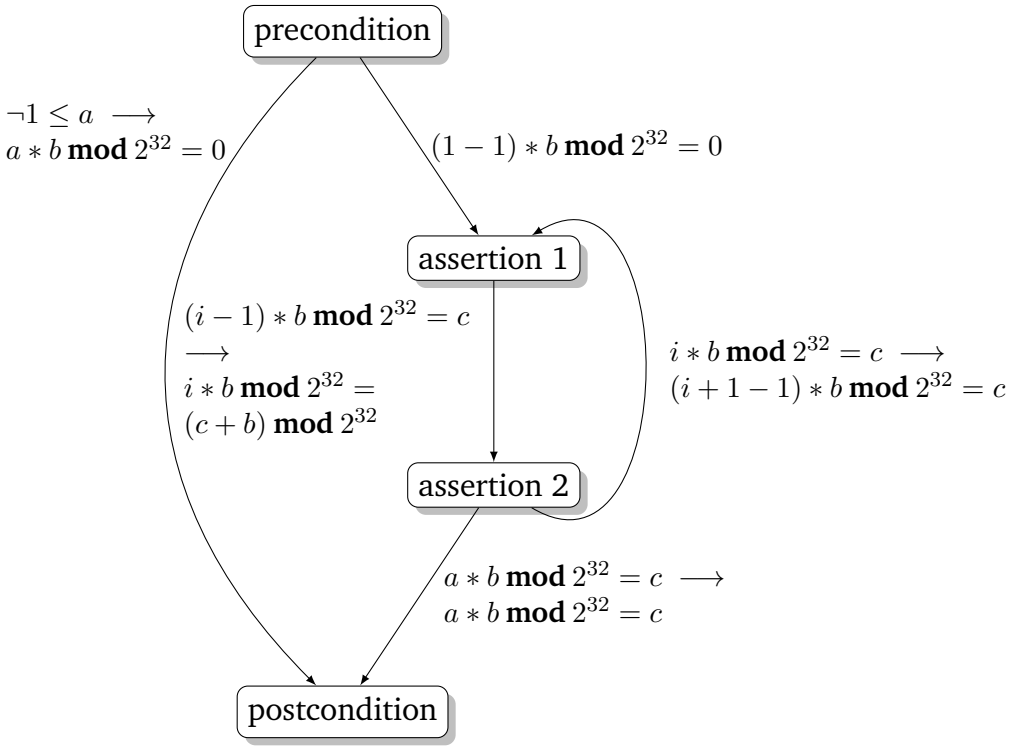


Figure 3.3: Control flow graph for procedure `Proc2`

maintain. The problem of having to prove the invariant several times can be avoided by rephrasing the above for-loop to

```
for I in T range L .. U loop
  --# assert P (I);
  S;
  --# assert P (I + 1)
end loop;
```

The VC for the path from the second assertion to the first assertion is trivial and can usually be proved automatically by the SPARK Simplifier, whereas the VC for the path from the first assertion to the second assertion actually expresses the fact that `S` preserves the invariant.

We illustrate this technique using the example package shown in Fig. 3.1. It contains two procedures `Proc1` and `Proc2`, both of which implement multiplication via addition. The procedures have the same specification, but in `Proc1`, only one **assert** statement is placed at the beginning of the loop, whereas `Proc2` employs the trick explained above. After applying the SPARK Simplifier to the VCs generated for `Proc1`, two very similar VCs

$$\texttt{loop__1__i * b mod } 2^{32} \texttt{ = ((loop__1__i - 1) * b mod } 2^{32} \texttt{ + b) mod } 2^{32}$$

and

$$\texttt{a * b mod } 2^{32} \texttt{ = ((a - 1) * b mod } 2^{32} \texttt{ + b) mod } 2^{32}$$

remain, whereas for `Proc2`, only the first of the above VCs remains. Why placing **assert** statements both at the beginning and at the end of the loop body simplifies the proof of the invariant should become obvious when looking at Fig. 3.2 and Fig. 3.3 showing the *control flow graphs* for `Proc1` and `Proc2`, respectively. The nodes in the graph correspond to cut points in the program, and the paths between the cut points are annotated with the corresponding VCs. To reduce the size of the graphs, we do not show nodes and edges corresponding to runtime checks. The VCs for the path bypassing the loop and for the path from the precondition to the (first) assertion are the same for both procedures. The graph for `Proc1` contains two VCs expressing that the invariant is preserved by the execution of the loop body: one for the path from the assertion to the assertion, and another one for the path from the assertion to the conclusion, which corresponds to the last iteration of the loop. The latter VC can be obtained from the former by simply replacing i by a. In contrast, the graph for `Proc2` contains only one such VC for the path from assertion 1 to assertion 2. The VC for the path from assertion 2 to assertion 1 is trivial, and so is the VC for the path

from assertion 2 to the postcondition, expressing that the loop invariant implies the postcondition when the loop has terminated.

$\langle proof \rangle$

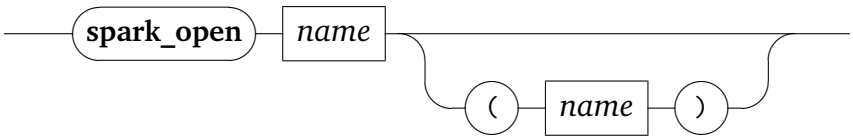
Chapter 4

HOL-SPARK Reference

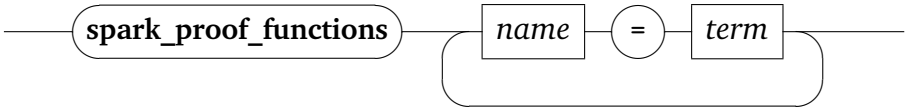
This section is intended as a quick reference for the HOL-SPARK verification environment. In §4.1, we give a summary of the commands provided by the HOL-SPARK, while §4.2 contains a description of how particular types of SPARK and FDL are modelled in Isabelle.

4.1 Commands

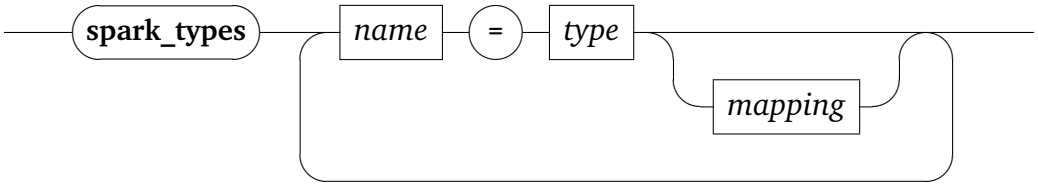
This section describes the syntax and effect of each of the commands provided by HOL-SPARK.



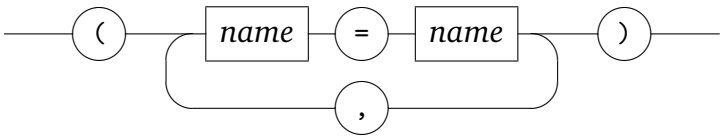
Opens a new SPARK verification environment and loads a `*.siv` file with VCs. Alternatively, `*.vcg` files can be loaded using **spark_open_vcg**. The corresponding `*.fdl` and `*.rls` files must reside in the same directory as the file given as an argument to the command. This command also generates records and datatypes for the types specified in the `*.fdl` file, unless they have already been associated with user-defined Isabelle types (see below). Since the full package name currently cannot be determined from the files generated by the SPARK Examiner, the command also allows to specify an optional package prefix in the format $p_1__\ldots__p_n$. When working with projects consisting of several packages, this is necessary in order for the verification environment to be able to map proof functions and types defined in Isabelle to their SPARK counterparts.



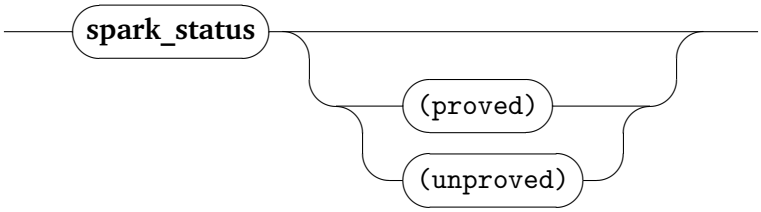
Associates a proof function with the given name to a term. The name should be the full name of the proof function as it appears in the `*.fdl` file, including the package prefix. This command can be used both inside and outside a verification environment. The latter variant is useful for introducing proof functions that are shared by several procedures or packages, whereas the former allows the given term to refer to the types generated by **spark_open** for record or enumeration types specified in the `*.fdl` file.



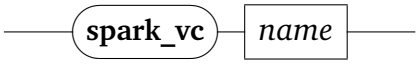
mapping



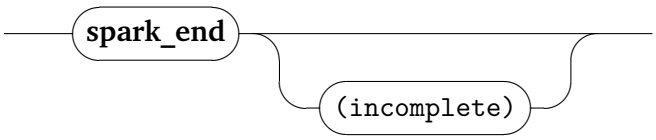
Associates a SPARK type with the given name with an Isabelle type. This command can only be used outside a verification environment. The given type must be either a record or a datatype, where the names of fields or constructors must either match those of the corresponding SPARK types (modulo casing), or a mapping from SPARK to Isabelle names has to be provided. This command is useful when having to define proof functions referring to record or enumeration types that are shared by several procedures or packages. First, the types required by the proof functions can be introduced using Isabelle's commands for defining records or datatypes. Having introduced the types, the proof functions can be defined in Isabelle. Finally, both the proof functions and the types can be associated with their SPARK counterparts.



Outputs the variables declared in the `*.fdl` file, the rules declared in the `*.rls` file, and all VCs, together with their status (proved, unproved). The output can be restricted to the proved or unproved VCs by giving the corresponding option to the command.



Initiates the proof of the VC with the given name. Similar to the standard **lemma** or **theorem** commands, this command must be followed by a sequence of proof commands. The command introduces the hypotheses `H1` ... `H`n, as well as the identifiers `?C1` ... `?C`m corresponding to the conclusions of the VC.



Closes the current verification environment. Unless the `incomplete` option is given, all VCs must have been proved, otherwise the command issues an error message. As a side effect, the command generates a proof review (`*.prv`) file to inform POGS of the proved VCs.

4.2 Types

The main types of FDL are integers, enumeration types, records, and arrays. In the following sections, we describe how these types are modelled in Isabelle.

4.2.1 Integers

The FDL type `integer` is modelled by the Isabelle type `int`. While the FDL `mod` operator behaves in the same way as its Isabelle counterpart, this is

```
sdiv_def:              a sdiv b = sgn a * sgn b * (|a| div |b|)
sdiv_minus_dividend:   - a sdiv b = - (a sdiv b)
sdiv_minus_divisor:    a sdiv - b = - (a sdiv b)
sdiv_pos_pos:          0 ≤ a ⟹ 0 ≤ b ⟹ a sdiv b = a div b
sdiv_pos_neg:          0 ≤ a ⟹ b < 0 ⟹ a sdiv b = - (a div - b)
sdiv_neg_pos:          a < 0 ⟹ 0 ≤ b ⟹ a sdiv b = - (- a div b)
sdiv_neg_neg:          a < 0 ⟹ b < 0 ⟹ a sdiv b = - a div - b
```

Figure 4.1: Characteristic properties of `sdiv`

```
AND_lower:    0 ≤ x ⟹ 0 ≤ x AND y
OR_lower:     0 ≤ x ⟹ 0 ≤ y ⟹ 0 ≤ x OR y
XOR_lower:    0 ≤ x ⟹ 0 ≤ y ⟹ 0 ≤ x XOR y
AND_upper1:   0 ≤ x ⟹ x AND y ≤ x
AND_upper2:   0 ≤ y ⟹ x AND y ≤ y
OR_upper:     0 ≤ x ⟹ x < 2^n ⟹ y < 2^n ⟹ x OR y < 2^n
XOR_upper:    0 ≤ x ⟹ x < 2^n ⟹ y < 2^n ⟹ x XOR y < 2^n
AND_mod:      x AND 2^n - 1 = x mod 2^n
```

Figure 4.2: Characteristic properties of bitwise operators

not the case for the `div` operator. As has already been mentioned in §2.2, the `div` operator of SPARK always truncates towards zero, whereas the `div` operator of Isabelle truncates towards minus infinity. Therefore, the FDL `div` operator is mapped to the `sdiv` operator in Isabelle. The characteristic theorems of `sdiv`, in particular those describing the relationship with the standard `div` operator, are shown in Fig. 4.1

The bitwise logical operators of SPARK and FDL are modelled by the operators `AND`, `OR` and `XOR` from Isabelle's `Word` library, all of which have type `int ⇒ int ⇒ int`. A list of properties of these operators that are useful in proofs about SPARK programs are shown in Fig. 4.2

4.2.2 Enumeration types

The FDL enumeration type

`type` t `=` $(e_1, e_2, \ldots, e_n)$`;`

is modelled by the Isabelle datatype

datatype t `=` e_1 `|` e_2 `|` `...` `|` e_n

```
range_pos:          range pos = {0..<int CARD('a)}
less_pos:           (x < y) = (pos x < pos y)
less_eq_pos:        (x ≤ y) = (pos x ≤ pos y)
val_def:            val = inv pos
succ_def:           succ x = val (pos x + 1)
pred_def:           pred x = val (pos x - 1)
first_el_def:       first_el = val 0
last_el_def:        last_el = val (int CARD('a) - 1)
inj_pos:            inj pos
val_pos:            val (pos x) = x
pos_val:            z ∈ range pos ⟹ pos (val z) = z
first_el_smallest:  first_el ≤ x
last_el_greatest:   x ≤ last_el
pos_succ:           x ≠ last_el ⟹ pos (succ x) = pos x + 1
pos_pred:           x ≠ first_el ⟹ pos (pred x) = pos x - 1
succ_val:           x ∈ range pos ⟹ succ (val x) = val (x + 1)
pred_val:           x ∈ range pos ⟹ pred (val x) = val (x - 1)
```

Figure 4.3: Generic properties of functions on enumeration types

The HOL-SPARK environment defines a type class `spark_enum` that captures the characteristic properties of all enumeration types. It provides the following polymorphic functions and constants for all types `'a` of this type class:

```
pos
val
succ
pred
first_el
last_el
```

In addition, `spark_enum` is a subclass of the `linorder` type class, which allows the comparison operators < and ≤ to be used on enumeration types. The polymorphic operations shown above enjoy a number of generic properties that hold for all enumeration types. These properties are listed in Fig. 4.3. Moreover, Fig. 4.4 shows a list of properties that are specific to each enumeration type t, such as the characteristic equations for `val` and `pos`.

t_val: `val` 0 `=` e_1
`val` 1 `=` e_2
$\vdots$
`val` $(n-1)$ `=` e_n

t_pos: pos $e_1 = 0$
pos $e_2 = 1$
$\vdots$
pos $e_n = n - 1$

t_card: `card(`t`)` `=` n
t_first_el: `first_el` `=` e_1
t_last_el: `last_el` `=` e_n

Figure 4.4: Type-specific properties of functions on enumeration types

4.2.3 Records

The FDL record type

`type` t `= record`
  f_1 `:` t_1`;`
  $\vdots$
  f_n `:` t_n
`end;`

is modelled by the Isabelle record type

record `t =`
  f_1 `::` t_1
  $\vdots$
  f_n `::` t_n

Records are constructed using the notation $(\!|f_1$ `=` v_1`,` $\ldots$`,` f_n `=` $v_n|\!)$, a field f_i of a record r is selected using the notation $f_i\ r$, and the fields f and f' of a record r can be updated using the notation r $(\!|f$ `:=` v`,` f' `:=` $v'|\!)$.

4.2.4 Arrays

The FDL array type

`type` t `= array [`t_1`,` $\ldots$`,` t_n`] of` u`;`

is modelled by the Isabelle function type $t_1 \times \cdots \times t_n \Rightarrow u$. Array updates are written as A`(`x_1 `:=` y_1`,` $\ldots$`,` x_n `:=` y_n`)`. To allow updating an array at a set of indices, HOL-SPARK provides the notation $\ldots$ `[:=]` $\ldots$, which can be combined with $\ldots$ `:=` $\ldots$ and has the properties

```
z ∈ xs ⟹ (f(xs [:=] y)) z = y
z ∉ xs ⟹ (f(xs [:=] y)) z = f z
f({x} [:=] y) = f(x := y)
```

Thus, we can write expressions like

```
A({0..9} [:=] 42, 15 := 99, {20..29} [:=] 0)
```

that would be cumbersome to write using single updates.

4.3 User-defined proof functions and types

To illustrate the interplay between the commands for introducing user-defined proof functions and types mentioned in §4.1, we now discuss a larger example involving the definition of proof functions on complex types. Assume we would like to define an array type, whose elements are records that themselves contain arrays. Moreover, assume we would like to initialize all array elements and record fields of type `Integer` in an array of this type with the value `0`. The specification of package `Complex_Types` containing the definition of the array type, which we call `Array_Type2`, is shown in Fig. 4.5. It also contains the declaration of a proof function `Initialized` that is used to express that the array has been initialized. The two other proof functions `Initialized2` and `Initialized3` are used to reason about the initialization of the inner array. Since the array types and proof functions may be used by several packages, such as the one shown in Fig. 4.6, it is advantageous to define the proof functions in a central theory that can be included by other theories containing proofs about packages using `Complex_Types`. We show this theory in Fig. 4.7. Since the proof functions refer to the enumeration and record types defined in `Complex_Types`, we need to define the Isabelle counterparts of these types using the **datatype** and **record** commands in order to be able to write down the definition of the proof functions. These types are linked to the corresponding SPARK types using the **spark_types** command. Note that we have to specify the full name of the SPARK functions including the package prefix. Using the logic of Isabelle, we can then define functions involving the enumeration and record types introduced above, and link them to the corresponding SPARK proof functions. It is important that the **definition** commands are preceeded by the **spark_types** command, since the definition of `initialized3` uses the `val` function for enumeration types that is only available once that `day` has been declared as a SPARK type.

```
package Complex_Types
is

   type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

   subtype Array_Index is Natural range 0 .. 9;

   type Array_Type1 is array (Array_Index, Day) of Integer;

   type Record_Type is
      record
         Field1 : Array_Type1;
         Field2 : Integer;
      end record;

   type Array_Type2 is array (Array_Index) of Record_Type;

   --# function Initialized
   --#   (A: Array_Type2; I : Natural) return Boolean;

   --# function Initialized2
   --#   (A: Array_Type1; I : Natural) return Boolean;

   --# function Initialized3
   --#   (A: Array_Type1; I : Natural; K : Natural) return Boolean;

end Complex_Types;
```

Figure 4.5: Nested array and record types

```
with Complex_Types;
--# inherit Complex_Types;

package Complex_Types_App
is

   procedure Initialize (A : in out Complex_Types.Array_Type2);
   --# derives A from A;
   --# post Complex_Types.Initialized (A, 10);

end Complex_Types_App;

package body Complex_Types_App
is

   procedure Initialize (A : in out Complex_Types.Array_Type2)
   is
   begin
      for I in Complex_Types.Array_Index
        --# assert Complex_Types.Initialized (A, I);
      loop
         for J in Complex_Types.Array_Index
           --# assert
           --#   Complex_Types.Initialized (A, I) and
           --#   Complex_Types.Initialized2 (A (I).Field1, J);
         loop
            for K in Complex_Types.Day
              --# assert
              --#   Complex_Types.Initialized (A, I) and
              --#   Complex_Types.Initialized2 (A (I).Field1, J) and
              --#   Complex_Types.Initialized3
              --#     (A (I).Field1, J, Complex_Types.Day'Pos (K));
            loop
               A (I).Field1 (J, K) := 0;
            end loop;
         end loop;
         A (I).Field2 := 0;
      end loop;
   end Initialize;

end Complex_Types_App;
```

Figure 4.6: Application of Complex_Types package

```
theory Complex_Types
imports "HOL-SPARK.SPARK"
begin

datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

record two_fields =
  Field1 :: "int × day ⇒ int"
  Field2 :: int

spark_types
  complex_types__day = day
  complex_types__record_type = two_fields

definition
  initialized3 :: "(int × day ⇒ int) ⇒ int ⇒ int ⇒ bool" where
  "initialized3 A i k = (∀j∈{0..<k}. A (i, val j) = 0)"

definition
  initialized2 :: "(int × day ⇒ int) ⇒ int ⇒ bool" where
  "initialized2 A i = (∀j∈{0..<i}. initialized3 A j 7)"

definition
  initialized :: "(int ⇒ two_fields) ⇒ int ⇒ bool" where
  "initialized A i = (∀j∈{0..<i}.
     initialized2 (Field1 (A j)) 10 ∧ Field2 (A j) = 0)"

spark_proof_functions
  complex_types__initialized = initialized
  complex_types__initialized2 = initialized2
  complex_types__initialized3 = initialized3
⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

end
```

Figure 4.7: Theory defining proof functions for complex types

Bibliography

[1] J. Barnes. *The* SPARK *Approach to Safety and Security*. Addison-Wesley, 2006.