SPARK 2014 User's Guide

リリース *18.0w*

AdaCore and Altran UK Ltd

1 月 18, 2017

Translation to Japanese by Mr. Masao Ito (NIL)

Contents

Chapter 1

SPARK を始めよう

新しいユーザが， SPARK ツールを動かしてみるためのとても簡単な例から始めることにします．具体的に理解できるように，小さな SPARK プログラム例を用いています．

注釈: このユーザガイドは， SPARK ツールの最新の開発バージョンに対する説明です．もし，SPARK GPL 版を利用している場合，ここに記述している特徴の幾つかは利用できません．ツールとともに受け取った SPARK ユーザガイドのバージョンを知るには，GPS と GNATbench IDE 上で， *Help* → *SPARK* を見るか， SPARK をインストールしたディレクトリから `share/doc/spark` をたどることで，バージョンが分かります．

SPARK ツールが，すでにインストールされていることが前提です．最低限必要なのは，次のツールです．

- SPARK Pro あるいは SPARK Discovery
- GPS 或いは the GNATbench plug-in (Eclipse 用）

SPARK Pro は， SPARK の完全なツールセットです． SPARK Discovery は，サブセットですが，このユーザガイドにある全ての解析を実行することができます．もちろん， SPARK Pro がより強力です． SPARK Pro と比較したとき， SPARK Discovery は：

- 自動証明器は，三つではなく一つになります
- 静的解析器 CodePeer と統合されていません
- 証明に失敗したときに反例が生成されません
- モジュラー演算ないしは，浮動小数点演算を用いたプログラムに対する証明支援が限定的です

（注）厳密に言えば，GPS は必須ではありません． SPARK では，全てのコマンドはコマンドラインから実行できるからです．また，Eclipse の GNATbench プラグインを用いることもできます．しかし，この節では，GPS を前提として説明します．もし，サポートサービスを受けていれば，ツールのインストールに関する更なる情報を入手することができます．GNAT Tracker の “Download” タブにある，”AdaCore Installation Procedures” を見るか，より詳細な情報が必要であれば AdaCore 社に，お問い合わせ下さい．

例で用いる主要ツールは， GNATprove と GPS です．まずはじめに，新しいデフォルトプロジェクトとして GPS を立ち上げましょう．メニューバーに，*SPARK* メニューがあることを確認してください．

注釈: SPARK 2005 もお使いの場合，このメニューは， *SPARK 2014* の下にあります．これは，SPARK 2005 ツールセットの *SPARK* と区別するためです．

GPS で新しいファイルを開きます．下記の短いプログラムを入力し， `diff.adb` として保存して下さい．

```
procedure Diff (X, Y : in Natural; Z : out Natural) with
  SPARK_Mode,
  Depends => (Z => (X, Y))
is
begin
   Z := X - X;
end Diff;
```

このプログラムは，X と Y の差を計算し，Z に保存します．このことは，Depends アスペクトに現れています．即ち，Z の値は，入力パラメータである X と Y の値に依存しているということです．もちろん，お気づきのようにこのプログラムにはバグがあります． SPARK_Mode aspect を利用することで，このコードは， SPARK であることが分かり，ツールによって分析することができます．GPS のメニューから， *SPARK → Examine File* を選んで下さい． GNATprove を，フロー解析モードで実行し，次のような報告を得ることができます：

```
diff.adb:1:20: warning: unused variable "Y"
diff.adb:1:36: info: initialization of "Z" proved
diff.adb:3:03: medium: missing dependency "null => Y"
diff.adb:3:24: medium: incorrect dependency "Z => Y"
```

これらの警告は，プログラムの契約（ Z は， X と Y から計算する）と，その実装（ Z は， X だけから計算され， Y は使用しない）の間に違いがあることを示しています．この場合は，契約が正しくプログラムが間違っています．それで，割り当て式を，次のように変更します． Z := X -Y; もう一度，解析を実行します．今度は，何の警告もエラーを示しません．

プログラムにフローエラーがないことが分かったので，実行時エラーを検出するために，立証モードでツールを実行します．GPS のメニューから， *SPARK → Prove File* を選び，結果ダイアログボックス中で， Execute をクリックします．GNATprove は，今度は，形式検証技術を用いて，プログラムには実行時エラーがないことを示そうと試みます．しかし，問題が見つかり，割り当て式を赤でハイライト表示します．次のように報告します．

```
diff2.adb:1:37: info: initialization of "Z" proved
diff2.adb:6:11: info: overflow check proved
diff2.adb:6:11: medium: range check might fail
```

このレポートは，ある Natural 番号から，別の Natural を引いたときに，結果が Natural となることをツールが示すことはできない，ということを示しています．これは，あまり驚くことであってほしくはない！

この改善策にはいくつもの方法がありますが，それは要求次第です．ここでは，引数 Z の型を Natural から Integer に変更します．修正の上，再度解析を実行すると，今度は， GNATprove は，エラーも警告も出力しません．全ての検査を通過したので，このコードを実行しても，例外が送出されることはない，と確信を持てます．

この短い例によって， SPARK ツールを用いて行う解析とはどういうことか，という感じを分かってもらえたらと思っています．

Chapter 2

はじめに

SPARK は，プログラム言語であり，検証ツール群です．高信頼ソフトウェア開発における必要となることを満たすように設計されています．SPARK 2014 は，Ada 2012 を元としていますが，次の意味でサブセットとなります．まず，検証を阻害する要素を取り除いています．また，モジュール化や形式検証を支援するために契約[1] やアスペクト[2] の仕組みを拡張しています．

新しいアスペクトを用いて，不完全なプログラムを分析することができます．また，抽象化 (abstraction) や洗練 (refinement) といった概念を支援し，詳細な静的解析が可能です．ここでは，情報のフロー解析や，仕様に対して実装が適合しているかの形式検証を含んでいます．

SPARK 2014 は，以前の SPARK 2005 と比較すると，より多機能で柔軟になっています．様々なアプリケーション領域や規格に適合できるように構成を変更することができます．サーバータイプの高信頼性システム（航空機管制システム）から，信頼性が要求される組み込みハードリアルタイムシステム（例えば，DO-178C Level A に適合する必要があるアビオニクスシステム）に用いることができます．

SPARK 2014 の最大の特徴は，証明と他の検証手段（テストなど）を一緒に利用できることです.（ある箇所では）単体テストの代わりに，ユニット毎の証明を行うことができます．DO-178C[3] や DO-333[4] といった規格では，この方法について記述が，形式手法に関する補足の中にあります．あるユニットを形式的に証明し，他のユニットではテストを通じて検証するといったことが可能です．

SPARK2014 では， GNAT に含まれる様々なツールを利用することができます．

- GNAT コンパイラ
- GPS 統合開発環境
- 単体テストハーネス生成のための GNATtest ツール
- 形式プログラム検証用の GNATprove ツール

この文書の残りの部分で，単に SPARK といえば，SPARK 2014 を指します．

この文書の他の部分は，以下の構成になっています．

- *GNATprove* **をインストールする** は，様々なプラットフォームへのインストール手順について記載しています．
- *SPARK* **コードを指定する** では，解析を行うために， SPARK で書かれたプログラム中の解析対象となる場所を指定するためのさまざまな方法を説明しています．
- *SPARK* **言語の概要** は， SPARK の概要について説明しています．

[1] 表明機構を用いることで，コンポーネントのふるまいを規定する方法．契約による設計．
[2] アスペクトは，Ada 言語のキーワード．”An aspect is a specifiable property of an entity. ” (ada 2012 reference manual)
[3] RTCA，Software Considerations in Airborne Systems and Equipment Certification
[4] RTCA，Formal Methods Supplement to DO-178C and DO-278A

- *GNATprove* **を用いた形式検証** は， GNATprove を用いた形式検証について説明しています.

Chapter 3

GNATprove をインストールする

一般に，SPARK をコンパイルするためには，最新のバージョンの GNAT ツールチェーン（Ada 2012 構文をサポートしているもの）をインストールする必要があります．ターゲットとする各プラットフォーム毎にツールチェーンをインストールする必要があります．例えば，あるツールチェーンは，自分の計算機用のネイティブコンパイルのためであり，別のものは，組み込みプラットフォームのためのクロスコンパイル用となります．

SPARK プログラムを解析するために，最初に，GPS（および任意で GNAT）をインストールし，次に，同じ場所に，GNATprove をインストールすることをお勧めします．GPS を用いないで，Eclipse 用の GNATbench プラグインをインストールすることもできます．このインストールには，Eclipse の一般的なインストール方法を利用します．GPS あるいは GNATbench の同一バージョンは，ネイティブ環境とクロスコンパイル環境の両方に利用できます．SPARK 解析も同様です．

GNATprove を別の場所にインストールする場合は，`GPR_PROJECT_PATH` 環境変数を変更する必要があります（GNAT をインストールしたならば）．ウィンドウズの場合は，環境変数パネル中で `GPR_PROJECT_PATH` を追加し，次のパスを設定します： `<GNAT install dir>/share/gpr` および， `<GNAT install dir>/share/gpr` ．これによって，SPARK は，GNAT とともにインストールしたライブラリプロジェクトを見つけることができるようになります．また，`<SPARK install dir>/lib/gnat` を追加します．これによって，GNAT は，SPARK とともにインストールされた補題ライブラリプロジェクト を見つけることができるようになります．Linux/Mac で Bourne シェルを使用している場合は，以下を設定して下さい．

```
export GPR_PROJECT_PATH=<GNAT install dir>/lib/gnat:<GNAT install dir>/share/gpr:
↪<SPARK install dir>/lib/gnat:$GPR_PROJECT_PATH
```

Linux/Mac で C シェルの場合は以下です:

```
setenv GPR_PROJECT_PATH <GNAT install dir>/lib/gnat:<GNAT install dir>/share/gpr:
↪<SPARK install dir>/lib/gnat:$GPR_PROJECT_PATH
```

GPS と GNATprove の詳細なインストール手順については，以下を参照下さい．

3.1 システム要求

形式検証は，複雑で時間を要します．利用可能な速度（CPU）と記憶域（RAM) が十分にあればあるほど GNATprove によって良いということになります．CPU コアあたり，2GB の RAM が推奨されます．より複雑な解析では，より多くのメモリが必要になります．大きなシステムに関して，GNATprove を動作させるための推奨構成は，Linux 64bit ないしは Windows 64bit OS で，少なくとも 8 コアと 16GB の RAM の構成となります．より遅い計算機で，小さなサブシステムを解析することは可能ですが，最低限 2.8Ghz の CPU と 2GB の RAM が必要となります．

さらに，もし， GNATprove と CodePeer を一緒に用いようとする場合（ GNATprove のスイッチは `--codepeer=on` ），10K SLOC のコードに対して約 1 GB のメモリが余分に必要になります．もし，300K SLOC を CodePeer で解析するとなると，64bit の構成で，少なくとも 30 GB の RAM が必要と云うことになります．もちろん，この値は，コードの複雑さによって変化します．コードが単純であれば，必要なメモリ量は少なくなりますし，とても複雑であれば，より多くのメモリを必要とします．

3.2 Windows へのインストール

まだであれば，最初に GPS インストーラを実行します．例えば，*gps-<version>-i686-pc-mingw32.exe* をダブルクリックし，指示に従います．

注釈: もし，GNAT Pro ではなく，GNAT GPL を使用する場合，GNAT GPL インストーラをお使い下さい．そうすれば，GPS がインストールされます．

同様に， GNATprove をインストーラを実行して下さい．例えば，spark-<version>-x86-windows-bin.exe をダブルクリックします．

パッケージをインストールするために，十分な権限を持っている必要があります（管理者権限か一般ユーザ権限かです．これは，全てのユーザ用にインストールしたいか，シングルユーザ用にインストールするかによって決まります）．

3.3 Linux/Mac にインストールする

まだであれば，GPS の tar 圧縮ファイルを展開し，インストールします．例えば，:

```
$ gzip -dc gps-<version>-<platform>-bin.tar.gz | tar xf -
$ cd gps-<version>-<platform>-bin
$ ./doinstall
```

次に，表示される指示に従います．

注釈: もし，もし，GNAT Pro ではなく，GNAT GPL を使用する場合，GNAT GPL インストーラをお使い下さい．そうすれば，GPS がインストールされます．

次に，SPARK tar 圧縮ファイルに対して，同様のことをして下さい．例えば．:

```
$ gzip -dc spark-<version>-<platform>-bin.tar.gz | tar xf -
$ cd spark-<version>-<platform>-bin
$ ./doinstall
```

指定した場所にパッケージをインストールために，十分な権限を持っている必要があります（例えば，/opt/spark 以下にインストールするためには，ルート権限が必要です）．

Chapter 4

SPARK コードを指定する

あるプログラムにおいて，一部が SPARK で書かれ（ SPARK の参照マニュアルにある規則に従っている），別のところでは完全な Ada 2012 で書かれるときがあります．このとき，プログラムないしはアスペクトにおける `SPARK_Mode` によって，どの部分が SPARK かを指定します（指定がない場合は，完全な Ada プログラムとみなします）．

この節では，pragma と `SPARK_Mode` アスペクトについて簡単に説明します．

GNATprove は，pragma ないしは `SPARK_Mode` アスペクトを用いて， SPARK と指定している場所に対してのみ，解析を行うことに注意して下さい．

4.1 SPARK コードと Ada コードの混在

Ada プログラムユニット [1] ないしは，他の構成要素が,「 SPARK 」であるというためには， SPARK 参照マニュアルに記述されている形式検証を可能とするために必要な制約に適合している必要があります．逆に，ある Ada プログラムユニットないしは，その構成要素が,「 SPARK ではない」のは，上記の要求に適合せず，結果として形式検証が実行できない時です．

Ada ユニットないしは，その構成要素を考えたときに， SPARK であるか，そうでないかは，次の 2 つの原則に従って，細かなレベルで組合わせることができます：

- SPARK コードは， SPARK 宣言のみを参照すること．しかし，Completion [2] を必要とする SPARK 宣言は，非 SPARK のボディ部となる場合がある．
- SPARK コードは， SPARK コードのみを含むべきである．ただし，含まれた SPARK 宣言に対応した非 SPARK Completion を含む場合がある．

もう少し詳細にいうと， 非 SPARK の completion となるえるのは，サブプログラム宣言・パッケージ宣言・タスク型宣言・保護型宣言・プライベート型宣言・プライベート拡張宣言・遅延定数宣言 [3] です．[厳密に言えば，パッケージのプライベート部分，タスク型，保護型は，上記の規則の目的からすると，completion と考えることができる．詳細については以下で記述します］

SPARK 宣言に対して，非 SPARK の completion が与えられるとき，非 SPARK completion が， （ SPARK の意味論という点から） SPARK 宣言を満足していることを示すのは，ユーザの責務になります．例えば， SPARK では，関数呼び出しで副作用を生じてはいけません．ある関数のボディ部が SPARK であるならば，種々の言語

[1] ユニットとは，サブプログラム・パッケージ・タスクユニット・保護ユニット（protected unit）・汎用体（generic unit）のいずれか．

[2] Ada 言語において，主として仕様部宣言に対するボディ部での宣言を指す（単純な例としては，あるパッケージの仕様部でサブプログラムが宣言されたとき，パッケージボディ部において，サブプログラムのボディ記述が必要になる．このボディ記述が Completion である）．ここではとりあえず原語を使用する．

[3] deferred constant declaration: この宣言は可視部分で行う．値は，プライベート部分で与えることができる．他言語で書かれたコードをインポートするときにも用いることができる．

規則に優先して適用されます．そうでなければ，関数ボディ部が規則に違反していないことを示すのはユーザの責務となります．そのような構成要素（特に pargma による仮定）と同様に，ルールに適合できていないと，一部・あるいは全ての SPARK に関連した解析（証明やフロー分析）は，無効になってしまいます．非 SPARK による completion が， SPARK で書かれた場合と意味的に（動的な意味論の点で）同等の completion 記述であるならば， それは，必要条件を満足しているといえます．

前述の要求に適合する混合プログラムにおいて， SPARK の意味論（特に，フロー分析や証明）は，明確に定義されています．この意味論は，100% SPARK で書かれたプログラムと同等になります．他の「混合」プログラムにおける意味論については，Ada の参照マニュアルを見て下さい．

パッケージ，タスク型，保護型，上記で示した仕様部と completion の区別は，実際には単純化しています．例えば，パッケージは，4 つに分割できます．2 つではありません．それは，可視部分，プライベート部分，ボディ部の宣言，そしてボディ部の一連の命令です．あるパッケージにおいて，0 .. 4 の数値 N が与えられた場合，パッケージにおいて，最初の N セクションまでは， SPARK であることができます．残りは違います．

例えば，次の組み合わせは典型的なものです．

- パッケージ仕様は， SPARK で書かれるが，パッケージボディ部は， SPARK ではない．
- パッケージ仕様部の可視部は SPARK であるが，プライベート部やボディ部は， SPARK ではない．
- パッケージ仕様部は， SPARK である．パッケージボディ部も，ほとんどは， SPARK である．しかし幾つかのサブプログラムボディ部は， SPARK ではない．
- SPARK で書かれたパッケージ仕様部がある．しかし，サブプログラムのボディ部は，他の言語からインポートしている．
- パッケージの仕様部は， SPARK と非 SPARK の混合で宣言されている．後者の宣言は， SPARK ではないクライアントユニットにとっては可視であり，使用することができる．

タスク型と保護型も上記のパッケージに似ています．しかし，4 つの部分ではなく，3 つに分かれます．ボディ部の命令列は含まれていません．

これらのパターンは，アプリケーションプログラムが，プログラムのサブセットに対してのみ形式検証を適用することを意図しています．これにより，形式検証とより伝統的なテストを（検証として）組合わせることができます．

4.2 プロジェクトファイルのセットアップ

プロジェクトファイルによって，プログラムのどの部分が SPARK かを粗くですが知ることができることができます．プロジェクトファイルのセットアップについて詳しく知るためには，次の章を見て下さい． **プロジェクトファイルの設定**.

4.2.1 デフォルトの SPARK_Mode を設定する

2 つのデフォルト（の利用の仕方）があります．

1. 構成 pragma 中で， `SPARK_Mode` に値が設定されていない場合です．このときは，プログラム中で明示的に `SPARK_Mode => On` を記述している箇所のみが， SPARK となります．サブプログラム中でわずかなユニットのみが SPARK である場合，このデフォルトの方法を推奨します．
2. 構成 pragma として， `SPARK_Mode => On` を設定している．この場合，明示的に `SPARK_Mode => Off` が記述されていない限り，全てのプログラムは， SPARK ということになります．ほとんどのプログラムが， SPARK で記述されている場合，このモードを推奨します．

構成 pragma として， `SPARK_Mode => On` の値を指定する方法を示します．

```
project My_Project is
   package Builder is
      for Global_Configuration_Pragmas use "spark.adc";
   end Builder;
end My_Project;
```

ここで， spark.adc が構成ファイルであり，少なくとも次の行を含んでいるとします.

```
pragma SPARK_Mode (On);
```

4.2.2 解析するファイルを指定する

デフォルトでは，プログラム中の全てのファイルを， GNATprove は解析します．もし，一部のプログラムだけが SPARK の場合，解析速度を向上させるために，該当するファイルのみを解析する方法もあります.

形式検証のモードに様々なプロジェクト属性に対して値を設定することで，解析対象ファイルを適切に指定することができます.

- Source_Dirs: ソースディクレトリの名前リスト
- Source_Files: ソースファイルの名前リスト
- Source_List_File: ソースファイルをリスト化しているファイルの名前

例えば，以下の様になります：

```
project My_Project is

  type Modes is ("Compile", "Analyze");
  Mode : Modes := External ("MODE", "Compile");

  case Mode is
     when "Compile" =>
        for Source_Dirs use (...);
     when "Analyze" =>
        for Source_Dirs use ("dir1", "dir2");
        for Source_Files use ("file1.ads", "file2.ads", "file1.adb", "file2.adb");
  end case;

end My_Project;
```

その上で， MODE 外部変数に対して，次のように値を指定した上で， GNATprove を呼び出します:

```
gnatprove -P my_project -XMODE=Analyze
```

4.2.3 解析対象からファイルを除外する

いま SPARK_Mode => On というデフォルト値を使用した場合，解析対象から幾つかのファイルを除かないといけない場合があります（例えば，非 SPARK コードを含んでいる，或いは，形式的に解析する必要がないコードを含んでいる場合です）.

形式検証のモードにおいて，様々なプロジェクト属性に対して異なる値を設定することで，解析対象から除外するファイルを適切に指定することができます.

- Excluded_Source_Dirs: 除外するソースディレクトリ名リスト
- Excluded_Source_Files: 除外するソースファイル名リスト

- Excluded_Source_List_File: 除外するソースフィル名をリスト化したファイルの名前

例えば，次のように記述します.

```
project My_Project is
   package Builder is
      for Global_Configuration_Pragmas use "spark.adc";
   end Builder;

  type Modes is ("Compile", "Analyze");
  Mode : Modes := External ("MODE", "Compile");

  case Mode is
     when "Compile" =>
        null;
     when "Analyze" =>
        for Excluded_Source_Files use ("file1.ads", "file1.adb", "file2.adb");
  end case;

end My_Project;
```

次に, MODE の値を外部変数として指定した上で， GNATprove を呼び出します:

```
gnatprove -P my_project -XMODE=Analyze
```

4.2.4 複数プロジェクトを利用する

デフォルトを SPARK_Mode => On とした上で，一部のソースファイルでは， SPARK_Mode を設定しないで解析すると，便利な場合があります．この場合，異なったデフォルト値を持つ 2 つのプロジェクトファイルを利用することができます．各ソースファイルは，どちらかのプロジェクトファイルのみに属するようにします．そうしても，あるプロジェクトのファイルは，他のプロジェクトのファイルを，プロジェクト間の制限的利用を示す limited with 節を用いて参照することができます．次のようになります.

```
limited with "project_b"
project My_Project_A is
   package Compiler is
      for Local_Configuration_Pragmas use "spark.adc";
   end Compiler;
   for Source_Files use ("file1.ads", "file2.ads", "file1.adb", "file2.adb");
end My_Project_A;
```

```
limited with "project_a"
project My_Project_B is
   for Source_Files use ("file3.ads", "file4.ads", "file3.adb", "file4.adb");
end My_Project_B;
```

ここで， spark.adc は，少なくとも次の行を含んでいる構成ファイルです.

```
pragma SPARK_Mode (On);
```

4.3 コード中で SPARK_Mode を用いる

プログラム中のどの部分が SPARK であるかを細かく指定するために，pragma ないしは SPARK_Mode アスペクトを用いることができます.

4.3.1 基本的な使い方

SPARK_Mode pragma は，次のように指定します.

```
pragma SPARK_Mode [ (On | Off) ]
```

例えば：

```
pragma SPARK_Mode (On);
package P is
```

SPARK_Mode アスペクトの場合は，次のようになります：

```
with SPARK_Mode => [ On | Off ]
```

例えば：

```
package P with
  SPARK_Mode => On
is
```

SPARK_Mode pragma ないしはアスペクトは，明示的に引数の指定のないまま用いると，デフォルトでは ON となります.

例えば:

```
package P is
   pragma SPARK_Mode;   -- ここでは暗黙的に ON です
```

或いは or

```
package P with
  SPARK_Mode   --   ここでは暗黙的に ON です
is
```

パッケージまたはサブプログラムが，トップレベルのパッケージであるか，ライブラリレベルパッケージ中で定義されているとき，ライブラリレベルと呼びます． SPARK_Mode pragma は，コード中の次の場所で使用することができます.

- ライブラリレベルパッケージ仕様部の先頭または前
- ライブラリレベルパッケージボディ部の先頭
- ライブラリレベルパッケージ仕様部の private キーワードの直後
- ライブラリレベルパッケージボディ部の begin キーワードの直後
- ライブラリレベルサブプログラム仕様の直後
- ライブラリレベルサブプログラムボディ部の先頭
- ライブラリレベルタスク仕様部の先頭
- ライブラリレベルタスクボディ部の先頭
- ライブラリタスク仕様部の private キーワードの直後
- ライブラリレベル保護仕様部の先頭
- ライブラリレベル保護ボディ部の先頭
- ライブラリレベル保護仕様部の private キーワードの直後

アスペクト SPARK_Mode は，コード中の以下の場所で利用可能です．

- ライブラリレベルのパッケージ仕様部あるいはボディ部
- ライブラリレベルのサブプログラム仕様部あるいはボディ部
- ライブラリレベルのタスク仕様部あるいはボディ部
- ライブラリレベルの保護仕様部あるいはボディ部

もし，サブプログラム・パッケージ・タスク・保護領域の仕様部／ボディ部で SPARK_Mode pragma ないしはアスペクトが指定されていない場合，宣言がある場所において有効な現在のモードを継承します．

［注］汎用体パッケージインスタンスは，インスタンス化される場所で宣言されていると考えます．従って，例えば，汎用体パッケージに， SPARK_Mode をマークすることはできませんし，サブプログラムのボディ部でインスタンス化することはできません．

4.3.2 一貫性規則

いったん， SPARK_Mode をオフにしたら，再度オンにはできない，というのが基本的な規則です．従って，次のルールが適用されます：

もし，あるサブプログラム仕様が， SPARK_Mode をオフにしたならば，ボディ部は， SPARK_Mode をオンにすることはできない．

パッケージには，4 つの部分があります．

1. パッケージの公開宣言
2. パッケージのプライベート部分
3. パッケージのボディ部
4. begin に続く，実行時準備処理 (elaboration) [4] コード

パッケージに関しては次のようになります．任意の箇所で明示的に SPARK_Mode をオフにしたときは，以降の場所では，SPARK_Mode をオンにすることができません．このとき，ボディ部において，再度 SPARK_Mode(Off) を必要とする場合もあることに注意して下さい．例えば，構成 pragma に，デフォルトとしてモードをオンにする SPARK_Mode (On) があるとします．あるパッケージの仕様部で， SPARK_Mode (Off) にしたとします．このとき，この pragma を，ボディ部で繰り返す必要があります．

タスク型と保護型も同様です．もし， SPARK_Mode が以下のある箇所でオフに設定されたならば，引き続く他の場所ではオンにすることはできません．

1. 仕様部
2. プライベート領域
3. ボディ部

このルールには例外があります． SPARK_Mode がオフである汎用体をインスタンス化したコードにおいて，汎用体中の SPARK_Mode の値は，このインスタンスでは無視されます．

4.3.3 利用例

選択したサブプログラムを検証する

もし，限られたわずかなサブプログラムが SPARK であり， SPARK_Mode をデフォルトでオンにすることに意味がないのならば， SPARK_Mode => On にする代わりに，該当するサブプログラムに対して，直接オンの指定を行います．例えば：

[4] Ada 言語においては，宣言がランタイム中に与える影響を処理する過程のこと．

```
package Selected_Subprograms is

   procedure Critical_Action with
     SPARK_Mode => On;

   procedure Sub_Action (X : out Boolean) with
     Post => X = True;

   procedure Non_Critical_Action;

end Selected_Subprograms;
```

[注] Sub_Action と Non_Critical_Action 手続きボディ部は解析されませんが，手続き Critical_Action のボディ部で Sub_Action を呼び出すことは正しいことです．このとき，Sub_Action の仕様部において，SPARK_Mode => On とする必要はありません．実際，GNATprove は，Sub_Action の仕様部が，SPARK であるとして検査します．

```
package body Selected_Subprograms is

   procedure Critical_Action with
     SPARK_Mode => On
   is
      --  this procedure body is analyzed
      X : Boolean;
   begin
      Sub_Action (X);
      pragma Assert (X = True);
   end Critical_Action;

   procedure Sub_Action (X : out Boolean) is
   begin
      --  this procedure body is not analyzed
      X := True;
   end Sub_Action;

   procedure Non_Critical_Action is
   begin
      --  this procedure body is not analyzed
      null;
   end Non_Critical_Action;

end Selected_Subprograms;
```

選択したユニットを検証する

もし，ある限られたわずかなユニットが SPARK であり，SPARK_Mode をデフォルトでオフにすることに意味があるのならば，SPARK_Mode => On にする代わりに，該当するユニットに対して，直接オンの指定を行います．例えば，

```
package Selected_Units with
  SPARK_Mode => On
is

   procedure Critical_Action;

   procedure Sub_Action (X : out Boolean) with
```

```
      Post => X = True;

   procedure Non_Critical_Action with
     SPARK_Mode => Off;

end Selected_Units;
```

[注] Sub_Action を，SPARK コード中で呼び出すことができます．なぜならば，ボディ部には，SPARK_Mode => Off とマークされていますが，その仕様部は SPARK だからです．逆に，仕様部が，SPARK_Mode => Off とマークされている Non_Critical_Action 手続きは，SPARK コード中で，呼び出すことができません．

```
package body Selected_Units with
  SPARK_Mode => On
is

   procedure Critical_Action is
      --  this procedure body is analyzed
      X : Boolean;
   begin
      Sub_Action (X);
      pragma Assert (X = True);
   end Critical_Action;

   procedure Sub_Action (X : out Boolean) with
     SPARK_Mode => Off
   is
   begin
      --  this procedure body is not analyzed
      X := True;
   end Sub_Action;

   procedure Non_Critical_Action with
     SPARK_Mode => Off
   is
   begin
      --  this procedure body is not analyzed
      null;
   end Non_Critical_Action;

end Selected_Units;
```

選択したユニットボディ部を除く

もし，あるユニットの仕様部が SPARK であり，ボディ部が SPARK ではない場合，仕様部は，SPARK_Mode => On とマークされ，ボディ部は，SPARK_Mode => Off とマークされます．こうすることで，SPARK で書かれたクライアントコードは，このユニットを使用することができます．もし，デフォルトで SPARK_Mode がオンであれば，そのユニットの仕様部でオンの指定を繰り返す必要はありません．

```
package Exclude_Unit_Body with
  SPARK_Mode => On
is

   type T is private;

   function Get_Value return Integer;
```

```
8
9     procedure Set_Value (V : Integer) with
10      Post => Get_Value = V;
11
12 private
13    pragma SPARK_Mode (Off);
14
15    --  the private part of the package spec is not analyzed
16
17    type T is access Integer;
18 end Exclude_Unit_Body;
```

仕様部のプライベート領域（物理的には仕様部ファイル中に存在することになりますが，論理的にはボディ部と同等になります）は， `SPARK_Mode (Off)` pragma をプライベート領域の最初に指定することで，同様に除外することができます．

```
1  package body Exclude_Unit_Body with
2    SPARK_Mode => Off
3  is
4     --  this package body is not analyzed
5
6     Value : T := new Integer;
7
8     function Get_Value return Integer is
9     begin
10       return Value.all;
11    end Get_Value;
12
13    procedure Set_Value (V : Integer) is
14    begin
15       Value.all := V;
16    end Set_Value;
17
18 end Exclude_Unit_Body;
```

この方式は，汎用体（generic）ユニットにおいても同様に機能します．次の場合にも成立します：`SPARK_Mode` がオンである場合，インスタンス化された汎用体のボディ部のみを除外する． `SPARK_Mode` がオフである場合，インスタンス化された汎用体の仕様部とボディ部を除外することができます．

```
1  generic
2     type T is private;
3  package Exclude_Generic_Unit_Body with
4    SPARK_Mode => On
5  is
6     procedure Process (X : in out T);
7  end Exclude_Generic_Unit_Body;
```

```
1  package body Exclude_Generic_Unit_Body with
2    SPARK_Mode => Off
3  is
4     --  this package body is not analyzed
5     procedure Process (X : in out T) is
6     begin
7        null;
8     end Process;
9  end Exclude_Generic_Unit_Body;
```

```
with Exclude_Generic_Unit_Body;
pragma Elaborate_All (Exclude_Generic_Unit_Body);

package Use_Generic with
  SPARK_Mode => On
is
   --  the spec of this generic instance is analyzed
   package G1 is new Exclude_Generic_Unit_Body (Integer);

   procedure Do_Nothing;

end Use_Generic;
```

```
package body Use_Generic with
  SPARK_Mode => Off
is
   type T is access Integer;

   --  this generic instance is not analyzed
   package G2 is new Exclude_Generic_Unit_Body (T);

   procedure Do_Nothing is
   begin
      null;
   end Do_Nothing;

end Use_Generic;
```

ユニットの選択した領域を取り除く

特定のサブプログラムとパッケージを除くほとんどのユニットが SPARK であるならば， `SPARK_Mode (On)` を設定することは合理的です．その上で， `SPARK_Mode => Off` を，非 SPARK 宣言に対して設定します．ここでは， `SPARK_Mode => On` を構成 pragma として指定することを考えます．

```
package Exclude_Selected_Parts is

   procedure Critical_Action;

   procedure Non_Critical_Action;

   package Non_Critical_Data with
     SPARK_Mode => Off
   is
      type T is access Integer;
      X : T;
      function Get_X return Integer;
   end Non_Critical_Data;

end Exclude_Selected_Parts;
```

手続き `Non_Critical_Action` が， SPARK コード中で呼ばれる場合があることに注意して下さい．ボディ部は， `SPARK_Mode => Off` とマークされていますが，仕様部は， SPARK コードだからです．

局所的パッケージ `Non_Critical_Data` は， `SPARK_Mode => Off` とマークされているので，非 SPARK 型，変数，サブプログラムを含むことができます．非 SPARK 宣言を集めたこういったパッケージを定義すると

便利かもしれません．そうしておけば，ユニット Exclude_Selected_Parts を SPARK_Mode => On としておくことができます．

```
package body Exclude_Selected_Parts is

   procedure Critical_Action is
   begin
      --  this procedure body is analyzed
      Non_Critical_Action;
   end Critical_Action;

   procedure Non_Critical_Action with
     SPARK_Mode => Off
   is
   begin
      --  this procedure body is not analyzed
      null;
   end Non_Critical_Action;

   package body Non_Critical_Data with
     SPARK_Mode => Off
   is
      --  this package body is not analyzed
      function Get_X return Integer is
      begin
         return X.all;
      end Get_X;
   end Non_Critical_Data;

end Exclude_Selected_Parts;
```

Chapter 5

SPARK 言語の概要

この章では， SPARK 言語の概要について説明します．実行と形式検証という面から，詳細は別途記載します．これは， SPARK 言語の参照マニュアルではありません．必要に応じて以下を参照して下さい．

- Ada 参照マニュアル（Ada 言語）
- SPARK 2014 リファレンスマニュアル（SPARK 言語に特有の要素）

GNAT が SPARK コードをどのようにコンパイルするかについては， GNAT 参照マニュアルを見て下さい．

SPARK は，Ada 言語の大きなサブセットと見なすことができ，追加のアスペクト／ pragma ／属性を持ちます．最新の SPARK 2014 は，以前のバージョンの SPARK と比べて，より大きなサブセットと見なすことができます．特に次の特徴を持ちます．

- 豊富な型（静的には範囲が定まらない副型，区別子付きレコード型，型述語）
- プログラムの構造化のためのより柔軟な特徴（関数，演算子多重定義，早期 return/exit，例外送出文）
- コード共有（汎用体，式関数）
- オブジェクト指向（タグ付き型，ディスパッチ）
- 並列動作（タスク，保護オブジェクト）

この章の残りでは，[Ada 2005] (或いは [Ada 2012]) によって， SPARK でサポートしている Ada の特徴が，どの Ada の版に基づいているかを示します．また， [Ravenscar] と標識がついている部分は， SPARK がサポートする Ravenscar プロファイルに基づく Ada の並列処理を示しています． [SPARK] が示しているのは，SPARK のみの特徴です． GNAT コンパイラと GNATprove 証明器は，ここに示している全ての特徴を支援します．

この節で示すコードの断片の幾つかは，SPARK ツールセットと一緒に配布している `gnatprove_by_example` の例にあります．ツールセットをインストールしたディレクトリ以下の `share/examples/spark` に見つけることができます．IDE(GPS ないしは GNATBench）からは，メニューアイテム *Help* → *SPARK* → *Examples* でアクセスすることができます．

5.1 言語上の制限

5.1.1 除外している Ada の特徴

形式検証を容易にするために， SPARK は，Ada 2012 に対して，幾つかの広域的な単純化を行っています．最も特徴的な単純化は以下のものです．

- アクセス型と割り当て子を利用することができません．ポインタが存在する場合，どのメモリが割り当てられ，どのメモリが解放されたかを追跡する必要があります．また，異なるメモリブロックの分離も考慮する必要があり，こういったことは，多数の労力なしに，精度よく行うことはできません．その代わり， SPARK では，豊富な汎用体データ構造を提供しています．詳細は，次を参照して下さい： Formal Containers Library
- 全ての式（関数呼び出しを含みます）は，副作用がありません．副作用を持つ関数を論理的に取り扱うのは非常に複雑で，非決定性評価が必要になる場合があります．これは，複合式中に含まれる式間で副作用の衝突を生じる場合があるためです．副作用を持つ関数は， SPARK では手続きとして書くべきです．
- 名前のエイリアス化を認めていません．エイリアス化は，思いがけない干渉を引き起こす可能性があります．局所的なはずの変数の値が，他の局所的名前変数を変更した結果，変わってしまう場合があるからです．エイリアス化を持つプログラムの形式検証は，正確さに欠け，多くの手作業を必要とします．詳しくは，次を参照下さい： **干渉しないこと**
- goto 文は認めていません．goto でループ文を作ることができ，その結果，形式検証においては特別な扱いが必要になるほか，正確にその場所を特定する必要があります．詳細については次を参照下さい．
- 被制御型 [1] は，利用できません．被制御型は，コンパイラによる暗黙的呼び出しを招くことになります．暗黙的呼び出しを形式検証しようとすると，形式検証ツールを利用するのに，ユーザが多くの手間を必要とすることになります．ツールが報告するときに，情報の元となるソースコードがないからです．
- 例外処理は除外しています．例外処理では，多数の手続き間の制御フローパスを作ることになります．例外処理があるプログラムの形式検証は，このパスに沿ってプロパティの確認をする必要があります．これは，多くの手作業なしに行うことはできません．しかし，例外の送出は可能です．(詳しくは，次を見て下さい： **例外の送出と他のエラー通知機構**)

上記に示した機能は， SPARK から除かれています．なぜならば，現時点で，形式検証のさまたげとなるからです．もし，形式検証技術が進化し，リストが更新されるならば，ここで示した制限の幾つかは緩和されるでしょう．形式的に検証することが可能だけれども，現時点では SPARK がサポートしていない特徴があります．例えば，access-to-subprogram 型 [2] です．

SPARK コード中で，これらの特徴を使用した場合， GNATprove が検出し，エラーとして報告します．これらの特徴を使用しているサブプログラムに対して，形式検証を行うことはできません．しかし， SPARK コードではありませんが，Ada 言語としては用いることができます．次も参照下さい： *SPARK* **コードを指定する**

5.1.2 Ada の特徴を部分的に解析する

SPARK では，Ada の強い型付けを強化しています．これは，より厳しい初期化ポリシー（ *Data Initialization Policy*) によるものです．また，ある種の入力データが不正であることを識別する手段を持ちません．結果として，以下の特徴は， SPARK で利用できますが， GNATprove では部分的にのみ解釈可能です．

- `Unchecked_Conversion` への呼び出しの結果は，返り値の型に対して正当な値であると見なします．
- 属性 `Valid` の評価結果は，常に真を返すと見なします．

このことを以下の例によって示します：

```
1  package Validity with
2    SPARK_Mode
3  is
4
5     procedure Convert (X : Integer; Y : out Float);
6
7  end Validity;
```

[1] 被制御型とは，自動的な初期化と再利用を可能とするタグ付き型（C++のコンストラクタとデストラクタに類似の機能を提供する）．
[2] サブプログラムの名前や宣言している場所を知ることなしに，呼び手はサブプログラムにアクセス可能とする．C 言語の関数ポインタに類似した機能

```
with Ada.Unchecked_Conversion;

package body Validity with
  SPARK_Mode
is

   function Int_To_Float is new Ada.Unchecked_Conversion (Integer, Float);

   procedure Convert (X : Integer; Y : out Float) is
   begin
      pragma Assert (X'Valid);
      Y := Int_To_Float (X);
      pragma Assert (Y'Valid);
   end Convert;

end Validity;
```

GNATprove は，両表明文を証明します．しかし，入力パラメータ `X` と `Unchecked_Conversion` 呼び出しの結果における `Valid` 属性の評価は，真を返すという前提に関して，警告を出力します：

```
validity.adb:11:22: info: assertion proved
validity.adb:11:22: warning: attribute Valid is assumed to return True
validity.adb:13:22: info: assertion proved
validity.adb:13:22: info: initialization of "Y" proved
validity.adb:13:22: warning: attribute Valid is assumed to return True
validity.ads:5:36: info: initialization of "Y" proved
```

5.1.3 データの初期化ポリシー

パラメータに対するモードおよびデータ依存性の契約（ *Data Dependencies* ）は， SPARK では，Ada よりもより厳格な意味を持っています．

- パラメータモード `in` (広域モードでは `Input`) は，このパラメータ（データ依存）によって示されるオブジェクトを，サブプログラムを呼び出す前に，完全に初期化しなければならないことを示しています．サブプログラム中で初期化することはできません．
- パラメータモード `out` (広域モードでは `Output`) は，このパラメータ（データ依存）によって示されるオブジェクトを，サブプログラムから制御が戻る前に，完全に初期化しなければならないことを示しています．初期化の前に，値を読み出すことはできません．
- パラメータモード `in out` (広域モードでは `In_Out`) は，このパラメータ（データ依存）によって示されるオブジェクトを，サブプログラムを呼び出す前に，完全に初期化しなければならないことを示しています．サブプログラム中で記述することも可能です．
- グローバルモードの `Proof_In` は，次のことを示しています．データ依存として示すオブジェクトは，サブプログラムを呼び出す前に，完全に初期化する必要があります．サブプログラム中で初期化されるべきではなく，契約と表明中でのみ読み出されるべきです．

以上により，全ての入力はサブプログラムへのエントリ点において，完全に初期化されているべきです．また，全ての出力はサブプログラムの出力点において，完全に初期化されているべきです．同様に，全てのオブジェクトは，読み出し（例えば，サブプログラム内部において）時に完全に初期化されているべきです．読み出しが初期化されているサポコンポーネントを提供されているレコード型サブコンポーネント（配列型サブコンポーネントは含まない）は例外です．

上記の規則の結果として，サブプログラム中で部分的に記述されているあるパラメータ（広域変数）は， `in out` (`In_Out`) とマークされるべきです．なぜならば，パラメータ（広域変数）の入力値は，サブプログラムから戻るときに， *read* となるからです．

GNATprove は，もしサブプログラムが，これまでに述べたデータ初期化ポリシーに従わない場合，検査メッセージを発行します．例えば，手続き Proc を見て下さい．各モードのパラメータと広域アイテムを持っています．

```
package Data_Initialization with
  SPARK_Mode
is
   type Data is record
      Val : Float;
      Num : Natural;
   end record;

   G1, G2, G3 : Data;

   procedure Proc
     (P1 : in     Data;
      P2 :    out Data;
      P3 : in out Data)
   with
     Global => (Input  => G1,
                Output => G2,
                In_Out => G3);

   procedure Call_Proc with
     Global => (Output => (G1, G2, G3));

end Data_Initialization;
```

手続き Proc は，その出力である P2 と G2 を完全に初期化すべきです．しかし，ここでは部分的な初期化のみです．同様に， Proc を呼び出す手続き Call_Proc は，呼び出しに先立ち，全ての Proc の入力を完全に初期化すべきです．しかし， G1 のみを完全に初期化しています．

```
package body Data_Initialization with
  SPARK_Mode
is

   procedure Proc
     (P1 : in     Data;
      P2 :    out Data;
      P3 : in out Data) is
   begin
      P2.Val := 0.0;
      G2.Num := 0;
      --  fail to completely initialize P2 and G2 before exit
   end Proc;

   procedure Call_Proc is
      X1, X2, X3 : Data;
   begin
      X1.Val := 0.0;
      X3.Num := 0;
      G1.Val := 0.0;
      G1.Num := 0;
      --  fail to completely initialize X1, X3 and G3 before call
      Proc (X1, X2, X3);
   end Call_Proc;

end Data_Initialization;
```

このプログラムでは， GNATprove は，6 つの重要な（"high"）検査メッセージを発行します．データ初期化ポリシーへの違反に関するものです．

```
data_initialization.adb:23:07: high: "G3.Num" is not an input in the Global contract␣
↪of subprogram "Call_Proc" at data_initialization.ads:20
data_initialization.adb:23:07: high: "G3.Num" is not initialized
data_initialization.adb:23:07: high: "G3.Val" is not an input in the Global contract␣
↪of subprogram "Call_Proc" at data_initialization.ads:20
data_initialization.adb:23:07: high: "G3.Val" is not initialized
data_initialization.adb:23:07: high: either make "G3.Num" an input in the Global␣
↪contract or initialize it before use
data_initialization.adb:23:07: high: either make "G3.Val" an input in the Global␣
↪contract or initialize it before use
data_initialization.adb:23:07: info: initialization of "G1.Num" proved
data_initialization.adb:23:07: info: initialization of "G1.Val" proved
data_initialization.adb:23:13: high: "X1.Num" is not initialized
data_initialization.adb:23:13: info: initialization of "X1.Val" proved
data_initialization.adb:23:17: warning: unused assignment to "X2"
data_initialization.adb:23:21: high: "X3.Val" is not initialized
data_initialization.adb:23:21: info: initialization of "X3.Num" proved
data_initialization.adb:23:21: warning: unused assignment to "X3"
data_initialization.ads:12:07: warning: unused variable "P1"
data_initialization.ads:13:07: high: "P2.Num" is not initialized in "Proc"
data_initialization.ads:13:07: info: initialization of "P2.Val" proved
data_initialization.ads:14:07: warning: "P3" is not modified, could be IN
data_initialization.ads:14:07: warning: unused variable "P3"
data_initialization.ads:16:27: low: unused global "G1"
data_initialization.ads:17:27: high: "G2.Val" is not an input in the Global contract␣
↪of subprogram "Proc" at line 11
data_initialization.ads:17:27: high: "G2.Val" is not initialized
data_initialization.ads:17:27: high: either make "G2.Val" an input in the Global␣
↪contract or initialize it before use
data_initialization.ads:17:27: info: initialization of "G2.Num" proved
data_initialization.ads:18:27: low: unused global "G3"
data_initialization.ads:18:27: warning: "G3" is not modified, could be INPUT
data_initialization.ads:21:28: info: initialization of "G1.Num" proved
data_initialization.ads:21:28: info: initialization of "G1.Val" proved
data_initialization.ads:21:32: info: initialization of "G2.Num" proved
data_initialization.ads:21:32: info: initialization of "G2.Val" proved
data_initialization.ads:21:36: info: initialization of "G3.Num" proved
data_initialization.ads:21:36: info: initialization of "G3.Val" proved
```

ユーザは，pragma `Annotate` を用いて，そういったメッセージを個々に正当化することができますが (詳細は次を参照下さい **検査メッセージを正当化する**)，呼び出されるサブコンポーネントが正しく初期化されることの責任は，ユーザ自身にあります． GNATprove は，データが呼び出される前に適切に初期化されているというプロパティを元にして証明するからです．

GNATprove は，使用していないパラメータ・広域アイテム・割り当てが存在しているという様々な警告も出力することに注意して下さい．これは，パラメータと広域モードに関する SPARK の厳密な解釈に基づくものです．

5.1.4 干渉しないこと

SPARK では，変数に対する割り当てによって，他の変数の値を変えることはできません． SPARK では，アクセス型（ポインタ）を禁止していることも，このことに寄与しています．また，パラメータや広域変数間で別名化を禁止しており，よい別名化（即ち，干渉を引き起こさない別名化）だけがよい別名化として利用可能です．

SPARK RM 6.4.2 に詳細記述している正確な規則をまとめると次のようになります：

- 二つの出力パラメータは，決して別名化することができません．
- 入力と出力パラメータは，入力パラメータが常にコピーで渡されるのでないならば，別名化するべきではありません．
- 出力パラメータは，サブプログラムによって参照される広域変数に別名化されるべきではありません．
- 入力パラメータは，もし常にコピーで渡されるのでないならば，サブプログラムによって参照される広域変数に別名化すべきではありません．

これらの規則は，別名化に制限を加えている Ada RM 6.4.1 の既存の規則を拡張しています．ちなみに，Ada では， *known to denote the same object* (Ada RM において正確に定義された記法) であるスカラー型のパラメータ間で問題ある（非良性の）別名化をしている手続きを呼び出すことができません．

例えば，次の例を参照して下さい．

```
package Aliasing with
  SPARK_Mode
is
   Glob : Integer;

   procedure Whatever (In_1, In_2 : Integer; Out_1, Out_2 : out Integer) with
     Global => Glob;

end Aliasing;
```

手続き `Whatever` は，以下の制約を満足する引数によってのみ呼び出すことができます．

1. 引数 `Out_1` と `Out_2` は，別名化すべきではありません．
2. 変数 `Glob` は，引数 `Out_1` や `Out_2` で渡されるべきではありません．

入力パラメータ `In_1` と `In_2` には，何の制約もないことに注意して下さい．この両者は，常にコピーが渡されるからです（スカラー型による良性のもの）．これらの入力パラメータが，レコード型や配列型の場合は，この限りではありません．

例えば，次は， `Whatever` を呼び出す場合の（Ada と SPARK の規則に従ったときの）正しい例と不正な例です．

```
with Aliasing; use Aliasing;

procedure Check_Param_Aliasing with
  SPARK_Mode
is
   X, Y, Z : Integer := 0;
begin
   Whatever (In_1 => X, In_2 => X, Out_1 => X, Out_2 => X);  --  illegal
   Whatever (In_1 => X, In_2 => X, Out_1 => X, Out_2 => Y);  --  correct
   Whatever (In_1 => X, In_2 => X, Out_1 => Y, Out_2 => X);  --  correct
   Whatever (In_1 => Y, In_2 => Z, Out_1 => X, Out_2 => X);  --  illegal
end Check_Param_Aliasing;
```

GNATprove は，正しく二つの不正な呼び出しを検知し，エラーを発行します (これは GNAT コンパイラでも同様です．両者とも Ada 言語の規則に従うからです）．

```
check_param_aliasing.adb:8:45: writable actual for "Out_1" overlaps with actual for
↪"Out_2"
check_param_aliasing.adb:11:45: writable actual for "Out_1" overlaps with actual for
↪"Out_2"
```

次は，他の例です.（SPARK 言語の規則に従った）手続き Whatever の正しい呼び出しと不正な呼び出しを示します.

```
with Aliasing; use Aliasing;

procedure Check_Aliasing with
  SPARK_Mode
is
   X, Y, Z : Integer := 0;
begin
   Whatever (In_1 => X, In_2 => X, Out_1 => X, Out_2 => Glob);      --  incorrect
   Whatever (In_1 => X, In_2 => Y, Out_1 => Z, Out_2 => Glob);      --  incorrect
   Whatever (In_1 => Glob, In_2 => Glob, Out_1 => X, Out_2 => Y);   --  correct
end Check_Aliasing;
```

GNATprove は正しく 2 つの不正な呼び出しを検知し，高レベルの検査メッセージを出力します.

```
check_aliasing.adb:8:22: info: initialization of "X" proved
check_aliasing.adb:8:57: high: formal parameter "Out_2" and global "Glob" are aliased␣
↪(SPARK RM 6.4.2)
```

5.1.5 例外の送出と他のエラー通知機構

エラーを通知するために， SPARK では，例外の送出が認められています．しかし，回復処理ないしは，緩和処理のために例外処理を行うことは， SPARK の範囲外になります．典型的には，そういった例外処理コードは，完全な Ada におけるトップレベルのサブプログラムでなされるべきです．あるいは，実行中に例外が送出されたとき呼ばれる最後のハンドラに追加すべきです．これらのいずれも GNATprove では解析されません.

GNATprove は，例外の送出に対して特別な処理をします.

- フロー解析において， raise_statement に至るプログラムのパスは，サブプログラムの契約を検査するときには，考慮されません．この検査では，正常に終了するプログラムの実行のみを検査します.
- 証明では，不達となるプログラムの場所がないことを証明するために，各 raise_statement に対して，検査を行います.

複数のエラー伝達機構も，同様に扱います.

- 例外を送出する.
- X が，静的に False に等価な式であるかを示す pragma Assert (X)
- 出力を持たない No_Return アスペクトないしは pragma を持つ手続き呼び出し（もし，呼び出し自身がそのような手続き中にないならば，検査は，エラー送出手続きの最外殻への呼び出しに対してのみ検査を行う）

例として作成したサブプログラム Check_OK を考えます．パラメータ OK が False であれば，例外を送出します.

```
package Abnormal_Terminations with
  SPARK_Mode
is

   G1, G2 : Integer := 0;

   procedure Check_OK (OK : Boolean) with
     Global => (Output => G1),
     Pre    => OK;
```

```
10
11 end Abnormal_Terminations;
```

```
1  package body Abnormal_Terminations with
2    SPARK_Mode
3  is
4
5     procedure Check_OK (OK : Boolean) is
6     begin
7        if OK then
8           G1 := 1;
9        else
10          G2 := 1;
11          raise Program_Error;
12       end if;
13    end Check_OK;
14
15 end Abnormal_Terminations;
```

変数 G2 は，手続き Check_OK 中で，値の割り当てを行いますが，その割り当て後に， raise_statement が続くので， G2 は，手続き Check_OK が正常に終了するとき，値を割り当てられません．結果として Check_OK のデータ依存に G2 は含まれていません．データフロー解析中 GNATprove は，手続き Check_OK のボディ部は，仕様部で宣言されたデータ依存を実装しているかを検査します．

証明中， GNATprove は，11 行目の raise_statement には決して到達しないことを検査します．これは，パラメータ OK は手続きに入ったときには True であるという Check_OK の事前条件によって検査可能となっています．

```
abnormal_terminations.adb:11:10: info: raise statement proved unreachable
abnormal_terminations.ads:8:27: info: initialization of "G1" proved
```

GNATprove は， No_Return アスペクトあるいは pragma によってマークされた手続きが，制御を返さないことを，検査します．例外を送出するか，任意の入力に対して無限にループすることになります．

5.2 サブプログラム契約

SPARK プログラムの振る舞いの意図を記述するために，最も重要な特徴は，サブプログラムに「契約」を加えることができることです．ここでは，*subprogram* は手続き，関数，保護エントリを指します．この契約は，様々な付加的な部品を用いて作成します．

- 事前条件 *precondition* では，サブプログラムを呼び出すときに呼び手が守るべき制約を，アスペクト Pre によって示します．
- 事後条件 *postcondition* では，サブプログラムの機能的振る舞いを，アスペクト Post によって（部分的にないしは全て）規定します．
- 契約ケース *contract cases* では，サブプログラムの振る舞いを分割するために，アスペクト Contract_Cases を用います．これは，事前条件・事後条件を補ったり，置き換えることができます．
- データ依存では，アスペクト Global を用いて，サブプログラムによって読み書きさえる広域データを特定します．
- フロー依存では，アスペクト Depends を用いて，サブプログラム出力がどのようにサブプログラム入力に依存しているかを識別します．

ある検証のゴールに対して書かれる契約にはどのようなものがあり， GNATprove はどのようにデフォルト契約を生成するかは， *How to Write Subprogram Contracts* に詳細を記述しています．

サブプログラムにおける契約は，呼び出しが成功するためのふるまいを規定します．エラーの送出により終了する実行は， **例外の送出と他のエラー通知機構** に記述しているように，サブプログラム契約では，カバーすることができません．実行が正常に終了する場合，あるいは．何らかの出力を持つアスペクト `No_Return` がマークされているサブプログラムに対して，エラーなしに実行がループするならば，サブプログラム呼び出しは成功です（制御器のメインループのサブプログラムを非終了サブプログラムとして実装することは典型的なケースとなります）．

5.2.1 事前条件

[Ada 2012]

サブプログラムの事前条件とは，そのサブプログラムを呼び出すコードに制約を加えることです．通常，事前条件は，制約の論理積として記述することができます．なお，各制約は以下のカテゴリのいずれかになります．

- 禁止された値をパラメータから除外．例えば， `X /= 0` 或いは `Y not in Active_States`
- 取り得るパラメータ値の仕様．例えば， `X in 1 .. 10` 或いは `Y in Idle_States`
- パラメータ間で維持すべき値の関係．例えば， `(if Y in Active_State then Z /= Null_State)`
- 計算途中の状態を表現する広域変数の期待値．例えば， `Current_State in Active_States`
- サブプログラムを呼び出すときに，維持すべき広域状態に関する不変条件．例えば， `Is_Complete (State_Mapping)`
- サブプログラムを呼び出すときに，維持すべき広域状態と入力パラメータの関係．例えば， `X in Next_States (Global_Map,Y)`

プログラムが，表明とともにコンパイルされるとき（例えば， GNAT において `-gnata` スイッチとともに），サブプログラムの事前条件は，実行中サブプログラムが呼び出される都度検査されます．事前条件が成立しない場合，例外を送出します．全ての表明で，このことが必要なわけではありません．例えば，共通のイディオムとしては，次のように pragma `Assertion_Policy` を設定することで，バイナリ中で事前条件のみを検査可能にします（他の表明は検査しません）．

```
pragma Assertion_Policy (Pre => Check);
```

サブプログラムを， GNATprove で解析するとき，実行するコンテキストを限定するために事前条件を用います．これは，サブプログラムの実装が以下であることを証明するために，一般に必要なことです．

- サブプログラムには実行時エラーがないこと
- サブプログラムの事後条件が常に維持されていることを保証すること

特に，デフォルトの事前条件 `True` を，明示的な事前条件が存在しない場合， GNATprove が用います．しかし， GNATprove によって呼び手のコンテキストを解析することができない時，正確ではない可能性があります． GNATprove が呼び手を解析する時，呼び出し時点で保持している呼び出されるサブプログラムの事前条件を検査します．そして， GNATprove によって，サブプログラムの実装が解析されないときでも，呼び出しているコードを解析するために，サブプログラムに対して事前条件を付け加えることは必要です．

例えば， `Add_To_Total` 手続きを考えます．これは，パラメータ `Incr` が持つ値によって，広域カウンタである `Total` の値を増やします．実装において，整数のオーバーフローが発生しないことを保証するために，`Incr` は大き過ぎる値をとることはできません．これは，次のような事前条件で示すことができます．

```
procedure Add_To_Total (Incr : in Integer) with
  Pre => Incr >= 0 and then Total <= Integer'Last - Incr;
```

変数 `Total` の値が，非負であることを保証するために，次の条件 `Total >= 0` を事前条件に加えることもできます．

```
procedure Add_To_Total (Incr : in Integer) with
  Pre => Incr >= 0 and then Total in 0 .. Integer'Last - Incr;
```

全てのコンテキストで，実行時エラーがないことを保証するために， GNATprove は，事前条件を解析します．このために，特別な事前条件の書き方が必要になります．例えば，前述の `Add_To_Total` の事前条件は，短いブール演算子である `and then` を， `and` の代わりに用いています．手続きを呼び出したときに， `Incr` が負であることを確認しているので， `Integer'Last -Incr``がオーバフローすることはありません．別の言い方をすると， ``and then` の利用により，式 `Integer'Last -Incr` が評価される前に，事前条件の非成立を確認できるということになります．

注釈: 事前条件中で， `and` の代わりに短いブール演算子である `and then` を使用するのは，よい習慣です．事前条件中で， GNATprove が実行時エラーがないことを証明するために，必要な場合があります．

5.2.2 事後条件

[Ada 2012]

サブプログラムの事後条件には，部分的，或いは完全なサブプログラムの機能的振る舞いを記述します．通常，事後条件は，次のカテゴリのいずれかの特性の論理積として書くことができます．

- 取り得る関数の返値．特別の属性 `Result` を使用します．次が例です： `Get'Result in Active_States`
- 出力パラメータの取り得る値．例えば， `Y in Active_States`
- 出力パラメータ間の期待する関係．例えば， `if Success then Y /= Null_State`
- 入力と出力パラメータ間の期待する関係．特別な属性である `Old` を用います．例えば， `if Success then Y /= Y'Old`
- 計算状態の更新を示す広域変数の期待する値．例えば， `Current_State in Active_States`
- サブプログラムからの戻りで保持すべき広域変数の不変条件．例えば， `Is_Complete (State_Mapping)`
- 広域状態とサブプログラムからの戻りで保持すべき出力パラメータとの間にある関係．例えば， `X in Next_States (Global_Map,Y)`

プログラムを表明とともにコンパイルするとき（例えば GNAT であれば `-gnata` スイッチを使用する），サブプログラムの事後条件は，実行中にサブプログラムから戻るときは常に検査されます．事後条件が不成立だった場合，例外が送出されます．通常，事後条件はテスト中有効です．事後条件は，プログラムが意図したとおり振る舞っていることを確認する動的で検査可能な答え（oracle）を提供しているからです．最終的にバイナリを作るときは，効率化のために動作しないようにすることができます．

サブプログラムを， GNATprove で解析する時，サブプログラムの事後条件が不成立とならないということを検査します．この検証は，モジュール化されています： GNATprove は，サブプログラムの事前条件が持っている全ての呼び出しコンテキストを考慮します． GNATprove は，また，実行時エラーを生じないということを保証するために，他の表明と同様の事後条件の解析を行います．

例えば，手続き `Add_To_Total` を考えます．これは，パラメータ `Incr` が持つ値によって，広域カウンタである `Total` の値を増加させます．この意図した振る舞いは，事後条件として次のように書くことができます..

```
procedure Add_To_Total (Incr : in Integer) with
  Post => Total = Total'Old + Incr;
```

サブプログラムの事後条件は，そのサブプログラムの呼び出しを解析するために用います．特に，明示的な事後条件がないとき， GNATprove が使用するデフォルトの事後条件 `True` は，呼び手の特性を証明するために

十分に正確ではないかもしれません．サブプログラムが呼び手のコンテキストで実装されていないときは，そうなります．

再帰的サブプログラムや相互に再帰的なサブプログラムは，ここでは，明示的に非再帰的サブプログラムとして扱います．これらサブプログラムは，常に終了します（特性は， GNATprove によって，検証されません）．GNATprove は各再帰呼び出し時の事後条件を用いることによって，事後条件に違反がないことを検査します．

ブール値を返す関数に対しては，特別な注意が必要です．よくある誤りは，事後条件として，期待するブール値結果を書いてしまうことです．

```
function Total_Above_Threshold (Threshold : in Integer) return Boolean with
  Post => Total > Threshold;
```

正しい事後条件として，次を用います:

```
function Total_Above_Threshold (Threshold : in Integer) return Boolean with
  Post => Total_Above_Threshold'Result = Total > Threshold;
```

GNAT コンパイラと GNATprove は，意味的には正しいが，機能的には間違っている可能性のある事後条件に対して警告を発行します．

5.2.3 契約ケース

[SPARK]

サブプログラムが，異なった機能的振る舞いの決まった組を持っているのであれば，これら振る舞いを事後条件というより契約ケースとして記述するのが便利です．例えば，ある手続きの変種を考えます．手続き `Add_To_Total` は，広域カウンタ `Total` を，それが可能な場合にパラメータ値を与えることにより増加させるか，ある閾値でそれ以上は大きくならないとします．これら振る舞いは，契約ケースでは次のように定義することが可能です．

```
procedure Add_To_Total (Incr : in Integer) with
  Contract_Cases => (Total + Incr < Threshold  => Total = Total'Old + Incr,
                     Total + Incr >= Threshold => Total = Threshold);
```

各契約ケースは，ガード中で構成され， 結果は，シンボル => により分離されます．サブプログラムへのエントリで，ガートが `True` と評価できたとき，サブプログラムの終了時に，対応する対応する条件文は， `True` と評価されます．この契約ケースは，呼び出しに対して起動された（enabled）ということができます．正確に一つの契約ケースが，各呼び出しに対して起動されるべきです．或いは，契約ケースは，互いに素であり完備しているべきと言うことができます．

例えば， `Add_To_Total` の契約ケースは，サブプログラムは 2 つの異なるケースのみで呼び出されるべきと示しています．

- `Total` の値を増加させる入力は，厳密に与えられた閾値より小さくなくてはなりません．この場合，手続き `Add_To_Total` は， `Total` を入力値分増加します．
- `Total` に入力値を加えたときに，閾値を超えるならば，手続き `Add_To_Total` は， `Total` の値を閾値の値とします．

プログラムを表明とともにコンパイルするとき（例えば， GNAT では， `-gnata` スイッチを用いる），全てのカード条件は，サブプログラムへのエントリ時点で評価されます．正確にどれか一つが `True` であることを実行時に検査します．この選択された契約ケースに関して，サブプログラムから戻ってきた時の別の実行時検査があります．それは，サブプログラムから制御が戻ってきた時に，関連する結果が `True` と評価できるかの検査です．

サブプログラムを， GNATprove とともに解析するとき，契約ケースのうち常に一つだけが有効であり，そのケースは結果として失敗しないことを検査します．もし，サブプログラムが事前条件も持つ場合， GNATprove は，事前条件を満足する入力のみ検査します．そうでない場合は，全ての入力をチェックします．

上記に挙げた単純な例において，式の書き方には，等価な事後条件となる複数の書き方があります：

```
procedure Add_To_Total (Incr : in Integer) with
  Post => (if Total'Old + Incr < Threshold  then
             Total = Total'Old + Incr
           else
             Total = Threshold);

procedure Add_To_Total (Incr : in Integer) with
  Post => Total = (if Total'Old + Incr < Threshold then Total'Old + Incr else␣
↪Threshold);

procedure Add_To_Total (Incr : in Integer) with
  Post => Total = Integer'Min (Total'Old + Incr, Threshold);
```

一般的に，等価な事後条件は，書きづらく・読みづらくなります．契約ケースはまた自動的に検証するための方法を提供しています．これは，入力空間を特定のケースに対応して分割することです．多くのケースがある場合，事後条件中の単純な式を分割することは困難です．

ケースのうちガード条件の最後を `others` とすることができます．これは，それ以前のどのケースにも含まれないあらゆるケースを表しています．例えば， `Add_To_Total` の契約は次のように書くことができます：

```
procedure Add_To_Total (Incr : in Integer) with
  Contract_Cases => (Total + Incr < Threshold => Total = Total'Old + Incr,
                     others                   => Total = Threshold);
```

キーワード `others` をガード条件として用いるとき，検証（実行時や GNATprove によるもの）は必要ありません．契約ケースが全ての可能な入力範囲をカバーしているからです．契約ケースが互いに素な場合のみ，検査を行います．

5.2.4 データ依存

[SPARK]

サブプログラムのデータ依存（Data Dependencies）によって，サブプログラムが読み書き可能な広域データを指定します．パラメータに関する記述とともに用いることで，サブプログラムの完全な入力および出力を規定できます．パラメータと同様に，データ依存中で示される広域変数は，入力に対して `Input` モード，出力に対して `Output` モード，入力でありかつ出力でもある広域変数に対して `In_Out` と記述します．そして，最後に， `Proof_In` モードです．これは，契約ないしは表明中でのみ読まれる入力を定義します．例えば，広域カウンタ `Total` を増加させる手続き `Add_To_Total` のデータ依存は，次のようになります．

```
procedure Add_To_Total (Incr : in Integer) with
  Global => (In_Out => Total);
```

保護サブプログラムでは，保護オブジェクトは，サブプログラムの暗黙的パラメータと考えます：

- 保護関数の暗黙的 `in` モードパラメータ
- 保護手続きないしは保護エントリーの暗黙的 `in out` モードパラメータ

データ依存は，プログラムのコンパイルや実行時の振る舞いに何の影響も与えません．サブプログラムを GNATprove を用いて解析する時，サブプログラムの次の実装を検査します：

- データ依存に指定のある広域入力のみを読み出しいるか
- データ依存に指定のある広域出力のみに書き込んでいるか
- 入力ではない広域出力を常に完全に初期化しているか

GNATprove の解析に関するより詳しい内容については， **データの初期化ポリシー** を参照のこと．解析中，GNATprove は，呼び手を解析するために，呼ばれているコードの記載されたデータ依存を使用します．もし，データ依存が存在しない場合は，呼ばれているコードに対するデフォルトのデータ依存契約が生成されます．

サブプログラム上のデータ依存を記述することにより，様々な利点があります．また，ユーザがデータ依存を契約に追加するのには，様々な理由があります．

- GNATprove は，サブプログラムの実装が，広域データへの指定したアクセスを遵守しているかを自動的に検証します．
- GNATprove は，サブプログラムの呼び手のデータおよびフロー依存を解析するために，フロー解析を行うときに指定した契約を利用します．これは単に生成されたデータ依存よりも精度の高い（即ち間違った警告が少ない）解析が可能となります．
- GNATprove は，実行時エラーがないこと，およびサブプログラムの呼び出し側の機能的な契約を検査するために，証明中に指定した契約を用います．こうすることで，単に生成されたデータ依存を用いるよりもより精度の高い（即ち間違った警告が少ない）解析が可能となります．

データ依存が，サブプログラム上で指定されているとき，サブプログラムにおける全ての広域データの読みだしと，書き込みの全てを指定すべきです．もし，サブプログラムが，広域的入力も出力も持たない場合は， `null` データ依存を用いて，記述することができます．

```
function Get (X : T) return Integer with
  Global => null;
```

サブプログラムが，広域入力のみを持ち，広域出力を持たない場合， `Input` モードを用いて指定します：

```
function Get_Sum return Integer with
  Global => (Input => (X, Y, Z));
```

或いは，モードなしで記載しても同値になります．

```
function Get_Sum return Integer with
  Global => (X, Y, Z);
```

［注］所与のモードに対する広域入力あるいは広域出力のリストには括弧を用いること．

読み書きされる広域データは， `In_Out` モードとして記述されるべきです．入力と出力と分けてはいけません．例えば， `Add_To_Total` に対するデータ依存の記述は不正であり， GNATprove は，エラーとします．

```
procedure Add_To_Total (Incr : in Integer) with
  Global => (Input  => Total,
             Output => Total);  --  INCORRECT
```

サブプログラム中で，部分的に記載されている広域データも，出力とはせずに `In_Out` とすべきです．詳しくは次を参照下さい **データの初期化ポリシー**.

5.2.5 フロー依存

[SPARK]

サブプログラムのフロー依存では，サブプログラムの出力（出力パラメータと広域的出力）が入力（入力パラメータと広域入力）に如何に依存しているかを指定します．例えば，広域カウンタ `Total` の値を増加する手続き `Add_To_Total` のフロー依存は次のように規定できます：

```
procedure Add_To_Total (Incr : in Integer) with
  Depends => (Total => (Total, Incr));
```

上記のフロー依存は，次のように読むことができます.「広域変数 Total の出力値は，広域変数 Total とパラメータ Incr に依存している」

保護サブプログラムに関しては，保護オブジェクトをサブプログラムの暗黙的パラメータと考えることができ，保護ユニット（型あるいはオブジェクト）という名前を使って，フロー依存中で次のように宣言可能です.

- 保護関数の暗黙的 in モードのパラメータとして．フロー依存の右手側に記載します.
- 保護手続き或いは保護エントリの暗黙的 in out モードパラメータとして． フロー依存の左手側・右手側両方に記載できます.

フロー依存は，プログラムのコンパイルや実行時の振る舞いに何の影響も与えません．サブプログラムが，GNATprove で解析されるとき，サブプログラムの実装中で，フロー依存で規定したように，出力が入力に依存していることを検査します．その解析中， GNATprove は，呼び手を解析するために，呼ばれるコードに規定されたフロー依存を利用します．もしフロー依存の記述がない場合，呼ばれる側のコードには，デフォルトのフロー依存契約が，生成されます.

フロー依存がサブプログラムにおいて指定された時，入力から出力への全てのフローを記述する必要があります．特に，部分的に書かれているパラメータの出力値或いは広域的変数が，その入力値に依存する場合はそうです.（詳しくは， **データの初期化ポリシー** を参照下さい）

パラメータあるいは広域変数の出力値が，その入力値に依存するとき，関係するフロー依存は，短縮シンボル * を使用することができます．このシンボルによって，変数の出力値は，変数の入力値とリスト化された他の入力に依存しているということ示すことができます．例えば， Add_To_Total のフロー依存は，次のように指定でき，それは同値となります：

```
procedure Add_To_Total (Incr : in Integer) with
  Depends => (Total =>+ Incr);
```

出力値が入力値に依存しないときは，出力値は定数によって（再）初期化されているだけなので入力値には依存しないということを意味し，そのことを示すフロー依存では， null 入力リストを用いることができます：

```
procedure Init_Total with
  Depends => (Total => null);
```

5.2.6 状態抽象と契約

[SPARK]

これまでに説明してきたサブプログラム契約では，直接に広域変数を扱ってきました．多くの場合そうすることができません．広域変数は，他のユニットで定義されていたり，直接見ることができないからです（パッケージ仕様のプライベート領域で定義されているか，パッケージ実装いおいて定義されているからです）．その場合は，契約における不可視の広域データを示すために， SPARK における抽象状態の表記を用いることができます.

状態抽象と依存

もし，手続き Add_To_Total によって値を増加する広域変数 Total がパッケージの実装で定義され，クライアントパッケージ中の手続き Cash_Tickets が， Add_To_Total を呼び出しているとします． Total を定義しているパッケージ Account は， Total を示す抽象状態 State を定義することができます． それを， Cash_Tickets のデータ・フロー依存中で，用いることができます.

```
procedure Cash_Tickets (Tickets : Ticket_Array) with
  Global  => (Output => Account.State),
  Depends => (Account.State => Tickets);
```

広域変数 Total は，ユニット Account のクライアントからは不可視になるので， Account の仕様部の可視領域においても不可視になります．それ故， Account における外部から可視のサブプログラムは，そのデータ・フロー依存中で，抽象状態 State を使う必要があります．例えば：

```
procedure Init_Total with
  Global  => (Output => State),
  Depends => (State => null);

procedure Add_To_Total (Incr : in Integer) with
  Global  => (In_Out => State),
  Depends => (State =>+ Incr);
```

次に， Init_Total と Add_To_Total の実装は，それぞれ Refined_Global と Refined_Depends によって導入した洗練したデータおよびフロー依存を定義することができます．この手続き中で，具体的な変数により，サブプログラムに対する正確な依存関係を与えます．

```
procedure Init_Total with
  Refined_Global  => (Output => Total),
  Refined_Depends => (Total => null)
is
begin
   Total := 0;
end Init_Total;

procedure Add_To_Total (Incr : in Integer) with
  Refined_Global  => (In_Out => Total),
  Refined_Depends => (Total =>+ Incr)
is
begin
   Total := Total + Incr;
end Add_To_Total;
```

ここで，洗練された依存性は， State を Total によって置き換えたときの抽象的依存と同様です．しかし，常にそうとは限りません．特に抽象状態が，複数の具体的な変数に置き換えられた場合は，異なります． GNATprove は次をチェックします．

- 各抽象広域 input が，具体的広域入力が示している，少なくとも一つの構成物を持っていること．
- 各抽象広域 in_out が，入力モードで指定す構成物の少なくとも一つを持っており，出力モードの一つ（或いは，in_out モードの少なくとも一つの構成物）を持っていること．
- 各抽象広域 output が，具体的広域出力によって示される全ての構成物を持っていること．
- 具体的フロー依存が，抽象フロー依存のサブセットであること．

GNATprove は，パッケージ Account の外部への呼び出しを解析する時， Init_Total と Add_To_Total の抽象契約（データとフロー依存）を用います．また，パッケージ Account の内部への呼び出しを解析する時 Init_Total と Add_To_Total のより正確で洗練した契約（即ち洗練したデータとフロー依存）を用います．

洗練した依存は，現在のユニット中で洗練された抽象状態を含んでいるデータと／またはフロー依存のサブプログラムおよびタスクの両方において指定することができます．

状態抽象と関数契約

もし，グローバル変数が，データ依存に対して可視状態にないとき，関数契約に対しても不可視ということになります．例えば，手続き Add_To_Total において，広域変数 Total が可視状態にない場合，関数 Add_To_Total において， **事前条件** および **事後条件** を表現することはできません．その代わり，表現する必要のある状態についてのプロパティを引き出すためのアクセッサとしての関数を定義し，契約に関して利用します．例えば：

```
function Get_Total return Integer;

procedure Add_To_Total (Incr : in Integer) with
  Pre  => Incr >= 0 and then Get_Total in 0 .. Integer'Last - Incr,
  Post => Get_Total = Get_Total'Old + Incr;
```

関数 Get_Total は，パッケージ Account のプライベート領域ないしは，実装として定義できます．また，通常の関数或いは関数式 の形式となります．例えば：

```
Total : Integer;

function Get_Total return Integer is (Total);
```

変数 Add_To_Total の実装に関して，洗練した事前条件や事後条件は必要としませんが， Refined_Post による洗練された事後条件を与えることは可能です．そして，より正確なサブプログラムの機能的振る舞いにを規定することができます．例えば，手続き Add_To_Total は，呼び出し毎に Call_Count カウンタの値を増加させることができ，洗練した事後条件中で表現することができます．

```
procedure Add_To_Total (Incr : in Integer) with
  Refined_Post => Total = Total'Old + Incr and Call_Count = Call_Count'Old + 1
is
   ...
end Add_To_Total;
```

洗練した事後条件は，ユニットが状態抽象を用いないときでさえ，或いは，サブプログラム宣言上で暗黙的に True 事後条件が用いられているときですら，サブプログラムの実装に与えることができます．

GNATprove は，パッケージ Account の外側で呼び出しを解析するとき， Add_To_Total の抽象契約（事前条件と事後条件）を用います．また， Account パッケージの内側で呼び出しを解析する時に， Add_To_Total のより正確な洗練した契約（事前条件と事後条件）を用いることができます．

Chapter 6

GNATprove を用いた形式検証

GNATprove ツールは， `gnatprove` と呼ばれる実行イメージとしてパッケージ化されています．一連の GNAT ツール中の他のツールと同様に， GNAT は，GNAT プロジェクトの構造に基づき， `.gpr` として定義されています．

検証中，実行時に解釈を行うのと全く同様に注釈を解釈するということは， GNATprove の大きな特徴です．特に，実行時意味論は，実行時検査の検証を含んでいます．この検証は， GNATprove によって静的に実行されます． GNATprove は，それ自身の期待される振る舞いの仕様およびコードとの関係に対する付加的な検証も行います．

6.1 GNATprove の実行方法

6.1.1 プロジェクトファイルの設定

基本的なプロジェクトの設定

まだ終わってなければ，GNAT プロジェクトファイル（*.gpr*）を作成します．このプロジェクトファイルについては， GNAT ユーザガイドの *GNAT Project Manager* の章を見て下さい．

GPS のプロジェクトウィザードを用いて，会話的にプロジェクトファイルを作成することもできます．メニューから *Project* → *New...* をたどります．特に最初のオプション（ *Single Project* ）を見て下さい．

もし，素早く始めたいのであれば，かつ `.ads/.adb` という小文字のファイル名を用いた標準的な名前規則に従い，ソースコードを単一ディレクトリ中に格納します．プロジェクトファイルは次のようになります：

```
project My_Project is
   for Source_Dirs use (".");
end My_Project;
```

これを `my_project.gpr` という名前で保存します．

コンパイルと検証に異なるスイッチを使う

場合によっては，GNAT や GNATprove に，異なったコンパイルレベルでのスイッチを渡したくなる場合があります．例えば，コンパイルのためだけに，同一プロジェクトファイル内で警告スイッチを用いるとします．この場合，コンパイルと検証で異なるスイッチを指定するために，シナリオ変数を利用することができます．

```
project My_Project is

  type Modes is ("Compile", "Analyze");
  Mode : Modes := External ("MODE", "Compile");

  package Compiler is
     case Mode is
        when "Compile" =>
           for Switches ("Ada") use ...
        when "Analyze" =>
           for Switches ("Ada") use ...
     end case;
  end Compiler;

end My_Project;
```

上記のプロジェクトでは, Compile デフォルトモードを用いてコンパイルします:

```
gprbuild -P my_project.gpr
```

形式検証は， Analyze モードを用い，次のようになります:

```
gnatprove -P my_project.gpr -XMODE=Analyze
```

6.1.2 コマンドラインで GNATprove を使う

GNATprove は，以下のようにコマンドラインで実行することができます:

```
gnatprove -P <project-file.gpr>
```

ここでは，もっともよく使われるものの概要を示しています． なお， GNATprove は，プロジェクトファイルがないと実行できないことにご注意ください．

GNATprove が解析するファイルを選択するために，共通に用いられる 3 つの方法があります．

- 全てを解析する:

  ```
  gnatprove -P <project-file.gpr> -U
  ```

  オプション -U を付けることで，プロジェクトツリー内の全てのプロジェクトの全てのユニットを解析します．まだ使用されていないユニットも含みます．

  これは，複雑なプロジェクトにおいて夜間に解析を実行したいときに通常用います．

- 指定したプロジェクトを解析する:

  ```
  gnatprove -P <project-file.gpr>
  ```

  指定プロジェクトの全てのメインユニットと，それらが依存している全てのユニット（これは再帰的に見つけます）を解析します．メインユニットが指定されていなければ，プロジェクト中の全てのファイルを解析します．

  特定の実行イメージのみの解析を行い時に用います．或いは，複雑なプロジェクト内部で，異なったオプションを持つ異なった実行イメージを解析したいときに用います．

- 単一ないしは複数ファイルを解析する:

```
gnatprove -P <project-file.gpr> [-u] FILES...
```

オプション -u を指定すると，指定したファイルのみを解析します．もし，このオプションが指定されないと，ファイルが依存している全てのユニットを再帰的に探し解析します．

これは，日々のコマンドラインでの解析や， GNATprove を IDE によって利用する場合に用います．

GNATprove は，2 つの異なった解析を行います．フロー解析と証明です．フロー解析は，データフローに関係したアスペクト（ Global, Depends, Abstract_State, Initializes およびこれらの洗練したバージョン）の正しさを検査し，また変数の初期化を検証します．証明は，実行時エラーがないこと，あるいは， Pre や Post アスペクトが示す表明の正しさを検証することです．スイッチ --mode=<mode> を用いることができ，モード（mode）には， check, check_all, flow, prove , all があります．任意の解析を選択することができます．

- モード check の場合， GNATprove は，プログラムが SPARK の制約を守っていることを部分的に検査します． check_all を使用する前に，このモードを使う利点は，フロー分析を必要としないので検査が高速になります．
- モード check_all の場合， GNATprove は，プログラムが SPARK の制約を守っていることを，全て検査します．関数に副作用がないといった，モード check では検査しない内容を含んでいます．即ち，モード check_all は，モード check を包含しています．
- モード flow では， GNATprove は，初期化していないデータをプログラムが入力とすることはない，規定したデータ依存あるいはフロー依存が実装において守られていることを検査します．モード flow は，モード check_all を包含しています．この段階は， *flow analysis* **（フロー解析）** と呼ばれます．
- モード prove では， GNATprove は，プログラムに実行時エラーがないこと，規定した関数契約が実装において遵守されていることを検査します．モード prove は，モード check_all を含んでおり，証明結果の十分性を保証するために，モード flow の機能の一つと同様に，初期化していないデータの読み込みがないことを検査します．この段階は， **証明（*proof*）** と呼ばれます．
- モード all はデフォルトのモードで， GNATprove はフロー解析と証明の双方を実行します．

オプション --limit-line= を使用することで，特定のファイルや Ada ファイル上の特定の行に証明を限定することができます．例えば，ファイル example.adb 上の 12 行目のみを証明したい場合，GNATprove の呼び出しにおいて， --limit-line=example.adb:12 を付加することができます．オプション --limit-subp= を使用することで，特定のファイル上の特定の行で宣言されたサブプログラムのみを証明の対象とすることができます．オプション -j は，並列計算と並列証明を指示します．

証明のふるまいに影響を与える多数のオプションがあります．内部的には，オプション --prover によって規定された証明器は，各検査ないしは表明で繰り返し呼ばれます．オプション --timeout を用いることで，各検査や表明を証明するために各証明器に許容する最大時間を，変更することができます．オプション --steps を使用することで（デフォルト：100），証明器が動作を止める前に実行可能な最大の推論ステップ数を設定することができます． steps オプションは，確実な結果が必要なときには用いるべきです．というのは，タイムアウトによる結果は，計算機の能力や，現在の計算機負荷によって変化するからです．オプション -jnnn では，nnn に示す値が最大コア数として並列計算します．オプション -j0 は，特別な意味を持ち，計算気が持つコア数を N としたときに，最大 N 並列で計算します．

注釈: プロジェクトがメインファイルを持つか，gnatprove に対して，あるファイルを開始点として指示する場合で，プロジェクト中の依存が線形である時（例えば，ユニット A は，ユニット B のみに依存していて，ユニット B は，ユニット C のみに依存しているといった場合）， -j スイッチを用いても，gnatprove は，ある時点では，一つのファイルのみを対象とします．この問題は，更に -U スイッチを用いることで避けることができます．

オプション --proof によっても，検証器に渡される検査方法もまた影響を受けます．デフォルトでは，証明器は，各検査毎あるいは表明毎に検査を実行します（モード per_check ）．これは，モード per_path を用いることで変わります．証明器は，検査における *path* 単位で検査を実行します．このオプションを用いると，

通常時間が多くかかるようになります．なぜならば，証明器は何度も実行することになるからです．しかし，良い証明の結果を得ることができる場合があります．最後は，モード `progressive` です．このモードでは，一つの検査で一度だけ証明器を実行しますが，証明できない場合には，分離したパス毎に，異なる方法を用いて，少しずつ検査を行います．

オプション `--proof` とともに設定された証明モードは，修飾子 `all` 或いは `lazy` を用いて拡張することが可能です．完全なスイッチの表現は，例えば次のようになります： `--proof=progressive:all` ． この修飾子を用いることで,（時間を節約するために）証明できなかった最初の場所で検査を停止するか，或いは,（通常証明されていない式を正確に特定するため，次に手動で証明するかもしれない）同じ検査に関係した他の式を証明するように，証明を継続するかを選択できます．前者は，完全な自動証明に最も適していますし，これがデフォルト値となっています．また明示的に，修飾子 `lazy` として選択することもできます．後者は，自動証明と手動証明の組み合わせに最も適しています．修飾子 `all` で選択することができます．

証明の速度と能力に影響を与える個々のスイッチを設定する代わりに，スイッチ `--level` を用いることもできます．レベルとは，既定義の証明レベルであり，最も高速なレベル 0（デフォルト値）から，最も強力なレベル 4 まであります．正確に言えば，各 `--level` の値は，これまでに示してきた他のスイッチを組み合わせたものと等価になります：

- `--level=0` は，次と等しい`--prover=cvc4 --proof=per_check --timeout=1`
- `--level=1` は，次と等しい`--prover=cvc4,z3,altergo --proof=per_check --timeout=1`
- `--level=2` は，次と等しい`--prover=cvc4,z3,altergo --proof=per_check --timeout=5`
- `--level=3` は，次と等しい`--prover=cvc4,z3,altergo --proof=progressive --timeout=5`
- `--level=4` は，次と等しい`--prover=cvc4,z3,altergo --proof=progressive --timeout=10`

もし `--level` と重要なスイッチ (`--prover` ， `--steps` または， `--proof`) が, 同時に設定された場合は, 後者が前者 `--level` の値を上書きします．

［注］ `--level` が同一でも，異なった計算機を使用した場合，結果が異なる場合があります．夜間のビルド（nightly builds）や，共有レポジトリでは， `--steps` or `--replay` スイッチを利用することを検討して下さい．証明に必要なステップ数は， `--report=statistics` オプションを付けて， GNATprove を実行することで得ることができます．

GNATprove は，静的解析ツール CodePeer を検査の証明に対する付加的な情報源として使用することを支援しています．このためには，コマンドラインで次を指定して下さい： `--codepeer=on` (詳細は次になります *CodePeer* **静的解析器を使う**)

デフォルトでは， GNATprove は，ユニットを単位として，変化していないファイルの再解析を行いません．このふるまいは，オプション `-f` により解除できます．

GNATprove は，ある項目を証明すると，結果をセッションファイルに保存します．ここには，要した時間と，証明までのステップが含まれます．この情報は，証明が正しかったということを確認するために，証明を再現するときに利用することができます． GNATprove を， `--replay` オプション付きで実行すれば，最後に証明したのと同じ証明器を使い，わずかに高い時間とステップ数で再現を試みます．このモードでは，ユーザが指示したステップ数と時間制約は無視されます．もし， `--prover` オプションがなければ， GNATprove は，全ての検査を試みます．そうでなければ，特定した証明器の一つによって証明された証明のみを再現します．全ての再現が成功すれば， GNATprove は，正常終了の場合と全く同様の出力を行います．もし再現が失敗すれば，関連する検査が，証明されなかったものととして報告されます．もし，関係する証明器が利用可能ではない（設定されていないサードパーティ製の証明器，あるいはユーザが `--prover` オプションを用いて他の証明器を選択した）場合は，証明を再現することができないという警告が発行されます．しかし，検査は依然として証明されたとマークされています．

デフォルトでは， GNATprove は,（Ada ないしは SPARK の規則違反により）エラーを検知した最初のユニットのところで停止します．オプション `-k` は， GNATprove が，複数ユニットで同種のエラーを発行するために用

います．もし，Ada の規則違反があった場合は， GNATprove は解析を試みようとはしません． SPARK の規則違反があった場合は， GNATprove は，検査フェーズのあと停止し，フロー解析と証明を試みません．

エラーを検知したとき（検査によるメッセージ出力は含みません）， GNATprove は，非ゼロの終了ステータスを返します．エラーが検知されなければ， GNATprove は，警告がありそのメッセージを出力したとしても，ゼロの終了ステータスを返します．

6.1.3 GNAT ターゲット実行時ディレクトリを使用する

もし，ターゲットコンパイラとして，GNAT を使用しており，用いているランタイムが， GNATprove のランタイムに含まれない場合，GNAT のインストール領域にある GNAT のランタイムを使用することができます．直接アクセスする，或いは SPARK のインストール領域にコピーすることによって可能となります．

ターゲットの GNAT ランタイムの場所を見つけるためには， `<target>-gnatls -v` コマンドを使用します． `--RTS` スイッチを使うと， `gnatls` を実行しているときに特定できます．

もし， GNATprove に渡される `--RTS` スイッチの引数が，正しい絶対ないしは相対ディクレトリ名ならば，GNATprove は，このディレクトリをランタイムディレクトリとして使用します．

正しくない場合， GNATprove は，既定義の場所で，ランタイムライブラリを探します．用いているランタイムの種類によって， 2つのケースがあります．

- 完全なランタイム

  例えば， `powerpc-vxworks-gnatmake` をビルドコマンドとして使い， `--RTS=kernel` とするならば，次を使えます：

```
powerpc-vxworks-gnatls -v --RTS=kernel | grep adalib
```

  `rts-kernel` ディレクトリを見つけ，このディレクトリを SPARK のインストール領域にコピーします．場所は次です． `<spark-install>/share/spark/runtimes` コピーするためには，例えば，*bash* 構文を使うと次になります:

```
cp -pr $(dirname $(powerpc-vxworks-gnatls -v --RTS=kernel | grep adalib)) \
  <spark-install>/share/spark/runtimes
```

  もし，プロジェクトファイル中でまだ指定してなければ，次を加えます：

```
package Builder is
   for Switches ("Ada") use ("--RTS=kernel");
end Builder;
```

  或いは，もし最近の GNAT と SPARK の版を使っているのであれば， *Runtime* プロジェクト属性経由で，ランタイムを指定できます：

```
for Runtime ("Ada") use "kernel";
```

- 構成可能なランタイム

  SPARK において，構成可能なランタイムを利用する最も単純な方法は， SPARK と，クロス GNAT コンパイラを同一のルートディレクトリにインストールすることです．

  その上で，プロジェクトファイル中に，Target と Runtime プロパティセットを記載すれば， GNATprove（バージョン 16.0.1 以降）は，ランタイムを自動的に見つけることができます．例えば：

```
for Target use "arm-eabi";
for Runtime ("Ada") use "ravenscar-sfp-stm32f4";
```

もし，上記の単純な解決策が使えない場合は，次のコマンドを使って，GNAT 構成可能ランタイムの場所を最初に見つける必要があります．

```
<target>-gnatls -v --RTS=<runtime> | grep adalib
```

これによって次へのパスが分かります `<runtime directory>/adalib`.

次の例で，arm-eabi ターゲットアーキテクチャ上の ravenscar-sfp-stm32f4 ランタイムライブラリを使うことを考えます．

```
arm-eabi-gnatls -v --RTS=ravenscar-sfp-stm32f4 | grep adalib
```

このコマンドによって，次のファイルへのパスが分かります． `<ravenscar-sfp-stm32f4 runtime>/adalib`

次に <ravenscar-sfp-stm32f4 runtime> ディレクトリを SPARK インストール領域の `<spark-prefix>/share/spark/runtimes` 以下にコピー（或いは，Unix においてはシンボリックリンクを作成）します．例えば， *bash* を用いると次のようになります．

```
cp -pr $(dirname $(arm-eabi-gnatls -v --RTS=ravenscar-sfp-stm32f4 | grep adalib))␣
↪\
  <spark-prefix>/share/spark/runtimes
```

もし，プロジェクトファイルで指定していなければ，次をプロジェクトファイルに追加します．

```
for Runtime ("Ada") use "ravenscar-sfp-stm32f4";
```

6.1.4 ターゲットアーキテクチャの指定と定義された実装のふるまい

SPARK プログラムでは，曖昧さがないことが保証されます．それゆえ，プロパティの形式検証が可能となります．しかし，幾つかのふるまい（例えば， `Size` 属性のような表現属性値）は，利用するコンパイラに依存する場合があります．デフォルトでは， GNATprove は，GNAT コンパイラと同じ選択をします． GNATprove は，また特別のスイッチを用いて他のコンパイラもサポートしています．

- `-gnateT` ターゲットの構成を指定するため
- `--pedantic` 定義された実装のあり得る振る舞いについての警告のため

［注］スイッチ `--pedantic` を用いても， GNATprove は，幾つかの定義された実装のふるまいを検出するのみです．

［注］|GNATprove| は，ベースタイプに対して，8bit の最小の倍数を常に選択します．Ada コンパイラにとっては，安全で保守的な選択となります．

ターゲットのパラメータ化

GNATprove は，コンパイルのターゲットは，そのコンパイラを実行しているホストと同じであると，デフォルトで仮定します．従って，ターゲットに依存する値，例えば，エンディアンや標準型のサイズやアライメントも同じであると考えます．もし， GNATprove を実行するホストとターゲットが異なる場合， GNATprove に対して，ターゲットを指定する必要があります．

［注］現在，プロジェクトファイル中の `Target` 属性は（何も出力されることなく）無視されます．

代わりに，以下をプロジェクトファイルに追加する必要があります:

```
project My_Project is
   [...]
   package Builder is
      for Global_Compilation_Switches ("Ada") use ("-gnateT=" & My_Project'Project_
↪Dir & "/target.atp");
   end Builder;
end My_Project;
```

ここで，target.atp は，プロジェクトファイル my_project.gpr と同じディレクトリに保存されており，パラメータ化したターゲットの情報を含んでいます．このファイルの書式は，GNAT ユーザガイドの -gnateT のスイッチ記述の箇所に記述されています．

パラメータ化したターゲット情報の目的は次になります．

- クロスコンパイルによって GNATprove が動作するホストとは異なるターゲットを記述するため．もし， GNAT がクロスコンパイラであれば，ターゲットのためにコンパイラを呼び出すときに -gnatet=target.atp スイッチを用いることで，構成ファイルが生成されます．このスイッチを用いない場合は，ターゲットのためのファイルは手動で作成する必要があります．
- ホストとターゲットが同一であるときでも， GNAT とは異なるコンパイラを使用している時には，そのパラメータを記述する必要があります．この場合，ターゲットファイルは，手動で作成する必要があります．

以下は，構成ファイルの例です．ここでは，PowerPC 750 プロセッサを持つベアボードで，ビッグエンディアンで構成しています:

```
Bits_BE                       1
Bits_Per_Unit                 8
Bits_Per_Word                32
Bytes_BE                      1
Char_Size                     8
Double_Float_Alignment        0
Double_Scalar_Alignment       0
Double_Size                  64
Float_Size                   32
Float_Words_BE                1
Int_Size                     32
Long_Double_Size             64
Long_Long_Size               64
Long_Size                    32
Maximum_Alignment            16
Max_Unaligned_Field          64
Pointer_Size                 32
Short_Enums                   0
Short_Size                   16
Strict_Alignment              1
System_Allocator_Alignment    8
Wchar_T_Size                 32
Words_BE                      1

float           6   I   32   32
double         15   I   64   64
long double    15   I   64   64
```

また，デフォルトでは， GNATprove は，ホストのランタイムライブラリを使用します．しかし，これは，クロスコンパイルをするときに，ターゲットにとって不適切かもしれません．スイッチ --RTS=dir を用いて，GNATprove を呼ぶことで，異なるランタイムライブラリを指定することができます．なお， dir は，デフォルトのランタイムライブラリの場所です．ランタイムライブラリの選択については， GNAT ユーザガイドのツー

ル gnatmake における --RTS スイッチの記述部分で説明しています.

括弧のついた算術演算

Ada においては，括弧のつかない算術演算は，コンパイラによって演算順序が変更になる場合があります．このため，計算に失敗する場合もありますし（例えばオーバフロー検査によって），逆に成功する場合もあります．デフォルトでは，GNATprove は，全ての式を左から右に評価します．GNAT も同様です．スイッチ --pedantic を用いると，計算順序が変更になった全ての演算に対して，警告が出力されます：

- それ自身が二項加算演算である二項加算演算（+,-）の被演算子
- それ自身が二項乗算演算である二項乗算演算（*, /, mod, rem）の被演算子

6.1.5 CodePeer 静的解析器を使う

注釈: CodePeer は，SPARK Pro 17 以上の一部として利用可能です．しかし，SPARK Discovery には含まれていません.

CodePeer は，静的解析器であり，AdaCore 社によって開発・商用化されました (http://www.adacore.com/codepeer). GNATprove では，検査における証明において追加の情報源として， CodePeer を用いることができます．このためには，コマンドラインオプションとして， --codepeer=on を使用します． CodePeer は，自動証明の前に，実行されます．もし，特定の検査に関して証明ができた場合， GNATprove は，別の検証器を用いて，再度この検査を繰り返そうとはしません.

GNATprove を実行したとき， CodePeer は，解析のために，事前条件を生成しようとはせず，ユーザが記述した事前条件のみに基づいて動作します． GNATprove とともに用いたときの CodePeer 解析は，失敗しそうな検査を証明することができないという点において，確実にふるまいます． CodePeer は，厳格で契約に基づく解析を行う SPARK よりも，一般的により多くの証明を行うことができる可能性があります：

1. CodePeer は，サブプログラムのデータ依存に対して十分な近似を生成します．これは，サブプログラムの実装とサブプログラムに関係しているコールグラフ（call-graph）に基づいています． CodePeer は，従って，ユーザが示すデータ依存の情報が粗すぎて他の方法では演繹できないプロパティを証明することが可能です.
2. CodePeer は，ループにおけるループ不変条件の十分な近似を生成します． CodePeer は，従って，不十分なループ不変条件あるいはループ不変条件がないときに，他の方法では演繹できないプロパティを証明することができます.

加えて， CodePeer は，固定小数点の乗算および除算のまるめに関して，GNAT コンパイラと同様に動作します．結果的に，GNAT でコンパイルしたコードの解析に対して，正確な結果を得ることができます．もし，ある固定小数点算術演算を行う他のコードが，GNAT 以外のコンパイラでコンパイルしており，そのコードが固定小数点の乗算および除算をしているならば， CodePeer のまるめ方法とは異なっているかもしれません．その場合は， --codepeer=on は使うべきではありません.

CodePeer 解析は，浮動小数点演算を用いているコードを解析するときに特に有効です．というのは， CodePeer は，浮動小数点演算における限界値を証明するときに，高速かつ正確に動作するからです.

6.1.6 GPS で GNATprove を実行する

GPS から GNATprove を実行できます． GNATprove がインストールされており，PATH 上に存在するならば，以下に関して *SPARK* メニューが使用可能になっています.

サブメニュー	アクション
Examine All	プロジェクト中の依存関係にある全てのメインとユニットに対して，フロー解析モードで， GNATprove を実行します.
Examine All Sources	プロジェクト中の全てのファイルに対して，フロー解析モードで， GNATprove を実行します.
Examine File	現在のユニットとそのボディ部および全てのサブユニットに対して，フロー解析モードで， GNATprove を実行します.
Prove All	プロジェクト中の依存関係にある全てのメインとユニットに対して， GNATprove を実行します.
Prove All Sources	プロジェクト中の全てのファイルに対して， GNATprove を実行します.
Prove File	現在のユニットとそのボディ部および全てのサブユニットに対して， GNATprove を実行します.
Show Report	GNATprove が生成したレポートファイルを表示します.
Clean Proofs	GNATprove が生成した全てのファイルを削除します.

三つの "Prove..." エントリは，プロジェクトファイルが示すモードで GNATprove を実行します．もし，モードが指定していなければ，デフォルトモードである "all" で実行します.

メニュー *SPARK → Examine/Prove All* は，プロジェクト中の全てのメインファイルおよび依存している全てのファイル（依存は再帰的に調べます）に対して， GNATprove を実行します．ルートプロジェクトおよびルートプロジェクトに含まれるプロジェクトのメインファイルが対象です．メニュー *SPARK → Examine/Prove All Sources* は，全てのプロジェクトの全てのファイルに対して， GNATprove を実行します．メインファイルを持っていない，或いは，他のプロジェクトを含んでいないプロジェクトの場合は，メニュー *SPARK → Examine/Prove All* と *SPARK → Examine/Prove All Sources* は同じになります.

メニュー項目のキーボードショートカットは，GPS の *Edit → Key Shortcuts* を用いて設定することができます.

注釈: サブメニューによって表示されるパネルにおいて行った変更は，セッションが変わっても引き継がれます．チェックボックスやスイッチの選択が，前回と同様で良いかは，注意して確認する必要があります.

Ada ファイルを編集するときに， GNATprove を，*SPARK* コンテキストメニューから実行することもできます．右クリックで，コンテキストメニューが表示されます.

サブメニュー	アクション
Examine File	現在のユニット，ボディ部，全てのサブユニットに対して，フロー解析モードで GNATprove を実行します.
Examine Subprogram	現在のサブプログラムに対してフロー解析モードで， GNATprove を実行します.
Prove File	現在のユニット，ボディ部，全てのサブユニットに対して， GNATprove を実行します.
Prove Subprogram	現在のサブプログラムに対して， GNATprove を実行します.
Prove Line	現在の行に対して， GNATprove を実行します.
Prove Check	現在の不合格となった条件に対して， GNATprove を実行します． GNATprove は，どの条件が不合格となったかを知るために，このオプションに対して少なくとも一回は動作します.

メニュー *Examine File* と *Prove File* を除いて，他のサブメニューはまた総称体の内部のコードに対しても適用可能です．この場合，関係するアクションは，プロジェクト中の総称体の全てのインスタンスに対して適用されます．例えば，もしある総称体が 2 度インスタンス化されている時，総称体の内側のサブプログラムにおいて， *Prove Subprogram* を選択することで，総称体のインスタンス中の関連する 2 つのサブプログラムに対する証明を行います.

メニュー *SPARK → Examine ...* は， GNATprove の解析のための種々の設定を行うパネルを開きます．このパネルにおいて，重要なのは解析モードの選択です．ここでは， `check` モード， `check_all` モード `flow` (デ

フォルト) から選択します.

メニュー *SPARK* → *Prove ...* を選択すると， GNATprove を用いて解析を行うために用いる様々なスイッチを設定するためのパネルが開きます．デフォルトでは，このパネルには基本的な幾つかの設定がなされているのみです．例えば，証明レベルに関する設定があります（詳しくは， **コマンドラインで *GNATprove* を使う** 中の `--level` を参照方）． *Edit* → *Preferences* → *SPARK* において SPARK に対する `User profile` を `Basic` から `Advanced` に変更すると，証明のためのより複雑なパネルが表示されます．ここには，より詳細なスイッチがあります.

GNATprove プロジェクトのスイッチは，パネル `GNATprove` で変更することができます (*Project* → *Edit Project Properties* → *Switches*).

フロー解析および証明に関する検査を行ったときに，あるサブプログラムの特定の経路が不合格となった場合，GNATprove は，ユーザに対して経路情報を生成する場合があります．次の操作によって，ユーザは GPS 上にこの経路を表示することができます．最初の方法は，不合格を示す証明メッセージの左にあるアイコンをクリックすることです．二番目の方法は，エディタ中で関係する行の左にあるアイコンをクリックすることです．同じアイコンを再度クリックすると，経路は再び見えなくなります.

証明を用いて検証を行う検査に対して， GNATprove は，ユーザに対して反例も表示する場合があります（参照： **反例を理解する** ）．次の操作によって，ユーザは GPS 上にこの反例を表示することができます．最初の方法は，不合格を示す証明メッセージの左にあるアイコンをクリックすることです．二番目の方法は，エディタ中で関係する行の左にあるアイコンをクリックすることです．同じアイコンを再度クリックすると，反例は再び見えなくなります.

6.1.7 GNATbench から GNATprove を実行する

GNATprove は，GNATbench から実行することができます． GNATprove がインストールされており，PATH 上にある場合， *SPARK* メニューが表示され，その中には次の項目があります.

サブメニュー	アクション
Examine All	プロジェクト中の全てのメインとそれらが依存する全てのユニットに対して，フロー解析モードで GNATprove を実行します.
Examine All Sources	プロジェクト中の全てのファイルに対して，フロー解析モードで GNATprove を実行します.
Examine File	現在のユニット，そのボディ部，全てのサブユニットに対して，フロー解析モードで GNATprove を実行します.
Prove All	プロジェクト中の全てのメインとそれらが依存する全てのユニットに対して，GNATprove を実行します.
Prove All Sources	プロジェクト中の全てのファイルに対して， GNATprove を実行します.
Prove File	現在のユニット，そのボディ部，全てのサブユニットに対して， GNATprove を実行します.
Show Report	GNATprove が生成したレポートファイルを表示します.
Clean Proofs	GNATprove が生成した全てのファイルを削除します.

プロジェクトファイルで指定したモードで，3 つの “Prove...” エントリを実行できます．もし，モードが指定されてなければ，デフォルトモードである “all” で，実行できます.

次のメニュー *SPARK* → *Examine/Prove All* を選択することで，プロジェクト中の全てのメインファイルに対して， GNATprove が動作します．また，メインファイルが依存している全てのファイルも再帰的に調べられ，対象となります．ここでの全てのメインファイルとは，ルートプロジェクトおよびルートプロジェクトに含まれる全てのプロジェクト中に存在するメインファイルです． *SPARK* → *Examine/Prove All Sources* は，全てのプロジェクト中の全てのファイルに対して， GNATprove を実行します．メインファイルも他のプロジェクトも含まないプロジェクトでは，メニュー *SPARK* → *Examine/Prove All* と *SPARK* → *Examine/Prove All Sources* は同等になります.

注釈: サブメニューにより表示されるパネル内で，ユーザが行った変更は，異なるセッションでも有効になり

ます．以前に追加されたチェックボックスやスイッチの値が，今回も適切か否かと云うことを注意する必要があります．

Ada ファイルを編集しているときは，右クリックにより表示されるメニュー *SPARK* からも， GNATprove を実行することができます．

サブメニュー	アクション
Examine File	現在のユニット，ボディ部，サブユニットに対して，フロー解析モードで GNATprove を実行します．
Examine Subprogram	現在のサブユニットに対して，フロー解析モードで GNATprove を実行します．
Prove File	現在のユニット，ボディ部，サブユニットに対して， GNATprove を実行します．
Prove Subprogram	現在のサブプログラムに対して， GNATprove を実行します．
Prove Line	現在の行に対して， GNATprove を実行します．

6.1.8 GNATprove と手動での証明

ある条件が妥当であるということを証明器が自動的に証明できなかった場合，手動により妥当性のための証明を試みることができます．

付録（Appendix）において，デフォルトである GNATprove とは異なった証明器を使う方法について，説明があります．

コマンドラインでの手動による証明

GNATprove が利用する証明器が，対話型として設定されている時，各解析条件ごとに次のようになる:

- 指定条件で初めて用いる場合．このとき，ファイル（指定した証明器に対する入力となる条件を含んでいる）がプロジェクト証明ディレクトリに作られます． GNATprove は，生成したファイル名とともに，この条件に関連したメッセージを出力します．条件を確認するために，ユーザは生成したファイルの編集が必要になる場合があります．
- 証明器を指定条件で一度用いており，編集可能なファイルが存在する場合．証明器は，このファイルとともに動作し，証明の成功・失敗を報告します．これはデフォルトの証明器と同様になります．

注釈: 手動による証明のためのファイルを作成し，ユーザが編集した場合，検証器をそのファイルに基づいて実行するためには，もう一度，同一の検証器を GNATprove で指定する必要があります．条件が証明され，結果が一度 why3 セッションに保存されれば， GNATprove は，条件が妥当であるかを知るために，再度検証器を指定する必要はありません．

GNATprove を用いた解析では， `--limit-line` オプションを用いることで，単一の条件に限定することが可能です:

```
gnatprove -P <project-file.gpr> --prover=<prover> --limit-line=<file>:<line>:<column>:
↪<check-kind>
```

ここで， `check-kind` は，失敗した場合に GNATprove が出力するメッセージから推測できる文字列です．以下の表を参照下さい．

Warning	Check kind
run-time checks	
divide by zero might fail	VC_DIVISION_CHECK

次のページに続く

TABLE 6.1 – 前のページからの続き

Warning	Check kind
array index check might fail	VC_INDEX_CHECK
overflow check might fail	VC_OVERFLOW_CHECK
range check might fail	VC_RANGE_CHECK
predicate check might fail	VC_PREDICATE_CHECK
predicate check might fail on default value	VC_PREDICATE_CHECK_ON_DEFAULT_VALUE
length check might fail	VC_LENGTH_CHECK
discriminant check might fail	VC_DISCRIMINANT_CHECK
tag check might fail	VC_TAG_CHECK
ceiling priority might not be in Interrupt_Priority	VC_CEILING_INTERRUPT
interrupt might be reserved	VC_INTERRUPT_RESERRED
ceiling priority protocol might not be respected	VC_CEILING_PRIORITY_PROTOCOL
task might terminate	VC_TASK_TERMINATION
assertions	
initial condition might fail	VC_INITIAL_CONDITION
default initial condition might fail	VC_DEFAULT_INITIAL_CONDITION
call to nonreturning subprogram might be executed	VC_PRECONDITION
precondition might fail	VC_PRECONDITION
precondition of main program might fail	VC_PRECONDITION_MAIN
postcondition might fail	VC_POSTCONDITION
refined postcondition might fail	VC_REFINED_POST
contract case might fail	VC_CONTRACT_CASE
contract cases might not be disjoint	VC_DISJOINT_CONTRACT_CASES
contract cases might not be complete	VC_COMPLETE_CONTRACT_CASES
loop invariant might fail in first iteration	VC_LOOP_INVARIANT_INIT
loop invariant might fail after first iteration	VC_LOOP_INVARIANT_PRESERV
loop variant might fail	VC_LOOP_VARIANT
assertion might fail	VC_ASSERT
exception might be raised	VC_RAISE
Liskov Substitution Principle	
precondition might be stronger than class-wide precondition	VC_WEAKER_PRE
precondition is stronger than the default class-wide precondition of True	VC_TRIVIAL_WEAKER_PRE
postcondition might be weaker than class-wide postcondition	VC_STRONGER_POST
class-wide precondition might be stronger than overridden one	VC_WEAKER_CLASSWIDE_PRE
class-wide postcondition might be weaker than overridden one	VC_STRONGER_CLASSWIDE_POST

GPS での手動による証明

GNATprove を証明モードで実行した後，ローケーションタブ中の検査メッセージ上で右クリックするか，単一の条件の検査で失敗した行を右クリックすることによって（即ち，選択行に関して GNATprove は出力中に一つの検査のみを含んでいます），メニュー *SPARK → Prove Check* を利用できるようになります.

ダイアログボックスの中の “Alternate prover (代替検証器)” では，Alt-Ergo とは異なる別の検証器を指定することができます．もし，別の検証器が，”interactive（会話的）” と指定され場合， *SPARK → Prove Check* を実行することで，GPS は，手動による証明のためのファイルを開き，検証器に関係するエディタを立ち上げます．このエディタは，代替検証器の構成の中で指定されているものです.

いったんエディタが閉じられると，GPS は， *SPARK → Prove Check* を再実行します．ユーザは，同じ代替検証器が以前と同様に指定されていることを確認してください．実行後，証明が失敗した場合，GPS は，再編集を要求します.

6.2 GNATprove の出力の見方

GNATprove には，2 種類の出力があります．一つは，標準出力ないしは IDE (GPS ないしは GNATbench）に表示されるものであり，もう一つは， `gnatprove.out` ファイルに出力されます．このファイルは，プロジェクトのオブジェクトディレクトリのサブディレクトリ `gnatprove` の中にあります．

6.2.1 解析結果サマリーテーブル

ファイル `gnatprove.out` の最初にあるサマリーテーブルは，プロジェクトの全ての検査に対して，検証した結果の概要を示しています．テーブルは例えば次のようになっています．:

```
-------------------------------------------------------------------------------------------
 ↪--------------------------
SPARK Analysis Results      Total        Flow   Interval                          ␣
 ↪Provers   Justified   Unproved
-------------------------------------------------------------------------------------------
 ↪--------------------------
Data Dependencies               .           .          .
  .           .          .
Flow Dependencies               .           .          .
  .           .          .
Initialization               2100        2079          .
  .           .         21
Non-Aliasing                    .           .          .
  .           .          .
Run-time Checks               596           .          .    480 (altergo  31%, CVC4 ␣
 ↪69%)           .         116
Assertions                      3           .          .      3 (altergo  33%, CVC4 ␣
 ↪67%)           .           .
Functional Contracts          323           .          .    168 (altergo  24%, CVC4 ␣
 ↪76%)           .         155
LSP Verification                .           .          .
  .           .          .
-------------------------------------------------------------------------------------------
 ↪--------------------------
Total                        3022  2079 (69%)           .                        651␣
 ↪(22%)           .   292 (9%)
```

サマリーテーブルの各行は，以下の意味を持ちます:

行記述	説明
Data Dependencies	**データ依存** とパラメータモードの検証
Flow Dependencies	**フロー依存** の検証
Initialization	**データの初期化ポリシー** の検証
Non-Aliasing	**干渉しないこと** の検証
Run-time Checks	実行時エラーがないことを検証（Verification of absence of run-time errors (AoRTE) (`Storage_Error` の送出を除く)）
Assertions	Assertion Pragmas の検証
Functional Contracts	機能的契約の検証 (**サブプログラム契約**, Package Contracts と Type Contracts を含む)
LSP Verification	Object Oriented Programming and Liskov Substitution Principle （オブジェクト指向プログラムと Liskov の代替原則）関連の検証

テーブルの各カラムの意味は次の通りです．

- `Total` カラムは，このカテゴリに含まれる総検査数を示します．

- `Flow` カラムは，フロー解析によって証明された検査の数を示します．
- `Interval` カラムは，副式[1] の型限界に基づき，単純な浮動小数点表現の静的境界解析による検査（オーバーフローと範囲検査）の検査数を記述します．
- `Provers` カラムは，自動ないしは手動による検証器によって証明された検査の数を記載しています．このカラムには，用いた検証器の情報および各検証器によって証明された検査の割合が記載されます．ある検査項目が複数の検証器によって証明される場合があることに注意して下さい．従って，ここでは数ではなく割合を用いています．最初に実行する検証器（コマンドラインスイッチ `--prover` によって定まります）が，一般的には最も多くの検証項目を証明することにも注意して下さい．各検証器は，他の検証器によってすでに証明された検査項目に関しては呼び出されないからです．検証器の（使用）率は，アルファベット順で示します．
- `Justified` カラムは，ユーザが提供した *Annotate Pragma* **を用いた直接的正当化** に対する検査項目の数を示します．
- 最後に， `Unproved` カラムは，証明も正当化もしていない検査項目の数を示します．

6.2.2 メッセージのカテゴリ

GNATprove のメッセージには幾つかの種類があります．エラー，警告，検査，情報メッセージです．

- 次の場合にはエラーを出力します． SPARK 或いは他言語の規約違反や，解析を継続することができない問題が発生した場合です．エラーを抑止することはできません．解析を進めるためには，エラー指摘箇所を修正する必要があります．
- 警告は，変数の未使用の値，意味のない値の割り当て等の疑わしい状況に対して発行されます．警告は，`"warning: "` という接頭辞が付きます． `pragma Warnings` pragma によって，抑止可能です．詳しくは， **警告を抑制する** を参照下さい．
- 検査メッセージは，プログラムの正しさに影響を与えるかもしれないコードが持つ潜在的な問題がある場合に，発行されます．例えば，初期化忘れ，実行時検査の潜在的な不合格，証明されていない表明などです．検査メッセージは，重大さに関する情報を含んでいます．重大さに従って，メッセージテキストには，次の接頭辞がつきます． `"low: "`, `"medium: "` , `"high: "` です．検査メッセージは警告のように抑制することができません．しかし， pragma `Annotate` を用いて，個々には各メッセージを正当化することができます．詳しくは次の節を参照して下さい **検査メッセージを正当化する**
- 情報メッセージは，ユーザに GNATprove がある構成要素に関して限界があることを通知します．或いは，GNATprove の出力を誤解することを防ぐために通知します．また， GNATprove の幾つかのモードにおいて証明された検査に関しての報告としても発行されます．

6.2.3 出力におけるモードの影響

GNATprove を，4 つの異なったモードで実行することができます．次のスイッチを指定します: `--mode=<mode>` ここで，可能なモードの値としては， `check`, `check_all`, `flow`, `prove`, `all` があります（詳細については，次を参照して下さい **コマンドラインで** *GNATprove* **を使う**）．出力は選択したモードに依存します．

モード `check` あるいはモード `check_all` を選択した場合， GNATprove は， `SPARK_Mode` が `On` である全てのコードにおいて， SPARK の制約に違反しているエラーメッセージのリストを標準出力に出力します．

モード `flow` とモード `prove` を選択した場合， この検査は最初のフェーズで実行されます．

モード `flow` では， GNATprove は，初期化されていないデータの読み込みの可能性，データ依存・フロー依存と仕様と実装の不整合，使用していない値の割り当てやリターン文の不足といった疑わしい状況に対して，標準出力にメッセージを出力します．これらのメッセージは，全てフロー解析に基づいて作成されます．

[1] いま，式が 3 * X / 4 だとすると，3 * X は副式の一つ．検査は，最終的な式の値ではなく，副式レベルでも行う

モード prove では， GNATprove は，初期化されていないデータの読み込みの可能性（フロー解析を使用する），潜在的な実行時エラーおよび特定の機能契約と実装との不整合（証明を使用する）に対するメッセージを標準出力に出力します．

モード all では， GNATprove は， flow モードおよび prove モード双方のメッセージを標準出力に出力します．

もし --report=all, --report=provers , --report=statistics のいずれかのスイッチが指定された場合， GNATprove は，追加で，証明された検査項目に対する情報メッセージを標準出力に出力します．

GNATprove は， gnatprove.out ファイル中に広域プロジェクト統計情報を出力します．この情報は，メニュー *SPARK* → *Show Report* を用いて，GPS 上に表示することが可能です．統計情報は以下になります．

- どのユニットを解析したか（フロー解析，証明，或いは双方）
- これらユニットのうちどのサブプログラムが解析されたか（フロー解析，証明，或いは双方）
- この解析の結果

6.2.4 メッセージの記述

この節では， GNATprove が出力する可能性のあるメッセージの違いをリスト化します．各メッセージは，ソースコード中の正確な場所を示します．例えば，もし，ソースファイル file.adb が下記のような割り算を持つ場合:

```
if X / Y > Z then ...
```

GNATprove は,（例えば）次を出力します:

```
file.adb:12:37: medium: divide by zero might fail
```

ここで，割り算記号 / の位置は，12 行目の 37 カラムであり正確な記述となっています．以下の最初のテーブルの説明を見て下さい．割り算検査は，除数がゼロではないことを検証します．つまり，このメッセージは， Y に関するものであり， GNATprove は，それがゼロとはなり得ないことを証明できません．以下の表中の説明は，ソースコード上の位置とともに解釈する必要があります．

以下の表における説明では，証明によって発行される検査メッセージの種類を示しています．

メッセージ種別	説明
run-time checks	
divide by zero	割り算・mod・rem 演算子の第二被演算子が，ゼロでないことを検査する [2]
index check	インデックスが配列の境界内であることを検査する
overflow check	算術演算が基本型の境界内であることを検査する
range check	値が，期待されるスカラーサブタイプの境界内であることを検査する
predicate check	値が，適用可能な型述語を満足しているかどうかを検査する [3]
predicate check on default value	型に対するデフォルト値が，適用可能な型述語を満足しているかを検査する
length check	配列は，適切な配列副型の配列長であるかを検査する．

次のページに続く

TABLE 6.2 – 前のページからの続き

メッセージ種別	説明
discriminant check	区別記録型（discriminant record）[4] の区別子（discriminant）が，適切な値を持っているかを検査する．変異記録型の場合，記録型のフィールドに対する単純なアクセスに対して生じる．しかし，区別子の固定値が必要となる場合もある [5]
tag check	タグ付きオブジェクトのタグが適切な値を持っているかの検査をする [6]
ceiling priority in Interrupt_Priority	Interrupt_Priority 中で，アスペクト Attach_Handler を持っている手続きを含む保護オブジェクト [7] に設定された上限優先度を検査する
interrupt is reserved	Attach_Handler に設定された割り込みが予約されていないことを検査する
ceiling priority protocol	上限優先度プロトコルの利用が適切かを検査する．即ち，タスクが保護操作を呼び出した時，タスクの有効な優先度が，保護オブジェクトの優先度より高くない (ARM Annex D.3)
task termination	タスクが終了しないことを検査で示す．これは Ravenscar [8] で必要とされている．
assertions	
initial condition	実行時準備処理 (elaboration) の後で，パッケージの初期状態が真であることを検査する．
default initial condition	型のデフォルトの初期状態が，その型のオブジェクトに対してデフォルトの初期化を行ったのちも True であることを検査する．
precondition	指定した呼び出しの事前条件アスペクトが True と評価できるかを検査する．
call to nonreturning subprogram	エラー時に呼び出されるサブプログラム呼び出しが不達であることを検査する．
precondition of main	メイン手続きの事前条件アスペクトが，実行時準備処理 (elaboration) ののちに True であることを検査する．
postcondition	サブプログラムの事後条件アスペクトが，True であることを検査する．
refined postcondition	サブプログラムの洗練事後条件 (refined postcondition) が，True であることを検査する．
contract case	サブプログラムの終端において，契約ケースが True であることを検査する．
disjoint contract cases	契約ケース aspect の各ケースが，全て互いに素であることを検査する
complete contract cases	契約ケース aspect の各ケースにより，事前条件アスペクトによって認められている状態空間がカバーされていることを検査する．
loop invariant in first iteration	ループにおける最初の繰り返しでループ不変条件が真となることを検査する．
loop invariant after first iteration	ループにおける初回に続く繰り返しで，ループ不変条件が真となることを検査する．
loop variant	与えられたループ変数が，ループの各繰り返しで，指定したとおり減少／増加することを検査する．これは，ループの終了に関係する．

次のページに続く

TABLE 6.2 – 前のページからの続き

メッセージ種別	説明
assertion	表明が真となることを検査する.
raised exception	エラーを送出する文には到達しないことを検査する.
Liskov Substitution Principles	[9]
precondition weaker than class-wide precondition	サブプログラムの事前条件 aspect が，クラスレベルの事前条件よりも弱い条件となっていることを検査する
precondition not True while class-wide precondition is True	もし，クラスレベルの事前条件が True であるならば，サブプログラムの事前条件 aspect が，True であることを検査する.
postcondition stronger than class-wide postcondition	サブプログラムの事後条件 aspect が，クラスレベルの事後条件よりも強い条件であることを検査する.
class-wide precondition weaker than overridden one	サブプログラムのクラス全体の事後条件 aspect が，オーバライドされたクラスレベルの事前条件よりも弱い条件であることを検査する.
class-wide postcondition stronger than overridden one	サブプログラムのクラス全体の事後条件が，オーバライドされたクラスレベルの事後条件よりも強い条件であることを検査する.

次のテーブルは，全てのフロー解析メッセージを示しています．クラスの記号の意味は次の通りです．E: エラー W: 警告 C: 検査

メッセージ種別	クラス	説明
aliasing	E	2 つの形式的あるいは広域的パラメータが別名化している.
function with side effects	E	副作用を持つ関数が検知された.
cannot depend on variable	E	特定の式（例えば，区別仕様やコンポーネント宣言）においては，変数に依存してはいけない.
missing global	E	フロー解析が，広域或いは初期化アスペクトで記載のない広域変数を検知した.
must be a global output	E	フロー解析は，in モードの広域変数の更新を検知した.
pragma Elaborate_All needed	E	リモート状態抽象が，パッケージの実行時準備処理中に用いられた．Elaborate_All がリモートのパッケージ対して必要となる.

次のページに続く

[2] mod と rem はともに剰余演算子．負数の振る舞いが異なる.
[3] 型述語とは以下のように，副型を規定するための述語表現．type Prime is new Positive with Predicate => (for all Divisor in 2 .. Prime / 2 => Prime mod Divisor /= 0);
[4] パラメータ化された記録型
[5] 区別記録型の特別な形式
[6] タグ付き型（tagged）．オブジェクト指向における継承に類似した仕組みを実現するための形式
[7] 保護型（protected type）は，排他的にデータにアクセスすることを保証する．複数タスクによるデータへのアクセス等で使用する
[8] Ravenscar profile は，セーフティクリティカルおよびリアルタイムシステムのために Ada タスキングのサブセットを定義したもの
[9] 「Liskov 代替原則」の Liskov は，抽象データ型を最初に実装した CLU の設計者として知られる Barbara Liskov 氏にちなむ.

TABLE 6.3 – 前のページからの続き

メッセージ種別	クラス	説明
export must not depend on Proof_In	E	Proof_In とマークされている定数に依存しているサブプログラムの出力を検知した.
class-wide mode must also be a class-wide mode of overridden subprogram	E	オーバライドしているサブプログラムの広域契約とオーバライドされているサブプログラムの広域契約間での不適合がある.
class-wide dependency is not class-wide dependency of overridden subprogram	E	オーバライドしているサブプログラムの依存契約とオーバライドされているサブプログラムの依存契約間に不適合がある.
volatile function	E	非 volatile 関数は，広域 volatile 変数を持たない場合がある.
tasking exclusivity	E	二つのタスクが，同一の保護オブジェクト或いは同一の保留オブジェクト (suspension object) に関してサスペンドしない.
tasking exclusivity	E	2 つのタスクが，同一の非同期オブジェクトに対して読み書きをしない.
missing dependency	C	依存関係にあるにもかかわらず，依存が示されていない.
dependency relation	C	出力パラメータないしは広域変数が，依存関係に示されていない.
missing null dependency	C	変数に対する NULL 依存の記述がない.
incorrect dependency	C	状態依存が十分に記述されていない.
not initialized	C	フロー解析によって，初期化されていない変数が見つかった.
initialization must not depend on something	C	誤った初期化アスペクトが検知された.
type is not fully initialized	C	デフォルトで初期化する指定をしている型が，初期化されていない.
needs to be a constituent of some state abstraction	C	何らかの状態抽象を通して示すべき構成要素を，フロー解析が検知した.
constant after elaboration	C	実行時準備処理後，定数であるべきオブジェクトは，実行時準備処理後にその値を変更してはいけないし，他のサブプログラムの出力とはなりえない.
is not modified	W	変数が in out モードで宣言されているにも関わらず，決して修正されないならば，in モードで宣言することができる.
unused assignment	W	フロー解析が，割り当て後，決して読み出されることのない変数への割り当てを検知した.
initialization has no effect	W	初期化済みだが，読み出されることのないオブジェクトを，フロー解析が検知した
this statement is never reached	W	この文は決して実行されない（デッドコード）.
statement has no effect	W	フロー解析は，プログラムに影響を与えない文を検知した.
unused initial value	W	out, in out パラメータ或いは広域変数に影響を与えない，in ないしは in out パラメータおよび広域変数を見つけた.
unused	W	広域的に或いは局色的に宣言された変数が使用されていない.
missing return	W	リターン文が関数に存在していない可能性がある.
no procedure exists that can initialize abstract state	W	フロー解析が初期化不能な状態抽象を検知した.
subprogram has no effect	W	出力をしないサブプログラムを検知した.

次のページに続く

TABLE 6.3 – 前のページからの続き

メッセージ種別	クラス	説明
volatile function	E	volatile 広域変数を持たない volatile 関数を volatile とする必要がない.

注釈: フロー解析が出力するメッセージには，エラーに区分されるものがあり，これは抑制したり正当化することができない.

6.2.5 反例を理解する

ある検査項目が証明されないとき， GNATprove が反例を生成する場合があります．反例は，2 つの部分から構成されます.

- サブプログラムに対するパス或いはパスの組
- パス上に現れる変数への値の割り当て

反例を見るためにもっとも良い方法は，GPS 上で不合格となった証明メッセージの左にあるアイコンをクリックすることです．或いは，エディタ上で関連する行の左側をクリックすることです（次を参照：*GPS で GNATprove を実行する*). GNATprove は，色でパスを表示します．これらの値を表示しているエディタ上で（ファイルではなく）挿入する行によってパス上の変数の値を表示することができます．例えば，手続き `Counterex` について考えます.

```
1  procedure Counterex (Cond : Boolean; In1, In2 : Integer; R : out Integer) with
2    SPARK_Mode,
3    Pre => In1 <= 25 and In2 <= 25
4  is
5  begin
6     R := 0;
7     if Cond then
8        R := R + In1;
9        if In1 < In2 then
10          R := R + In2;
11          pragma Assert (R < 42);
12       end if;
13    end if;
14 end Counterex;
```

入力パラメータ `Cond` が `True` であり，入力パラメータ `I1` と `I2` が大きすぎる値の場合，11 行の表明文は不合格となります. GNATprove が生成する反例は，GPS では以下のように表示されます．ここで，パス上でハイライト表示される各行の次に，前の行における変数の値を示した行が続きます.

counterex.adb

```
1  procedure Counterex (Cond : Boolean; In1, In2 : Integer; R : out Integer) with
   --  Cond = True and In1 = 17 and In2 = 25 and R = 0
2    SPARK_Mode,
3    Pre => In1 <= 25 and In2 <= 25
4  is
5  begin
6     R := 0;
      --  R = 0
7     if Cond then
8        R := R + In1;
         --  R = 17
9        if In1 < In2 then
10          R := R + In2;
            --  R = 42
11          pragma Assert (R < 42);
            --  R = 42
12       end if;
13    end if;
14 end Counterex;
```

GNATprove は，また説明付きで，不合格になった証明に対するメッセージを完成させます．即ち，反例に対して，検査済みの式で変数がどういう値をとるかを付け加えます．ここで，11 行目で GNATprove は，出力パラメータ R の値を示すメッセージを出力します．

```
counterex.adb:11:25: medium: assertion might fail, cannot prove R < 42 (e.g. when R =
↪42)
```

GNATprove が生成する反例が，いつもプログラムの適切な実行と対応しているとは限りません．

1. 契約やループ不変条件が不足しているとき，プロパティは，証明不能になります（詳しくは次の節をご覧下さい **証明できないプロパティを調査する** ）．反例は，不足している契約やループ不変条件を見つけるのに役立つ場合があります．例えば，手続き Double_In_Call の事後条件は，証明不可能です．なぜならば，その手続きが呼び出す関数 Double の事後条件（の制約）が弱いからです．また，手続き Double_In_Loop の事後条件は証明不可能です．なぜならば，その手続き中のループがループ不変条件を持たないからです．

```
1  package Counterex_Unprovable with
2    SPARK_Mode
3  is
4
5     type Int is new Integer range -100 .. 100;
6
7     function Double (X : Int) return Int with
8       Pre  => abs X <= 10,
9       Post => abs Double'Result <= 20;
10
11    procedure Double_In_Call (X : in out Int) with
12      Pre  => abs X <= 10,
13      Post => X = 2 * X'Old;
14
15    procedure Double_In_Loop (X : in out Int) with
16      Pre  => abs X <= 10,
17      Post => X = 2 * X'Old;
18
19 end Counterex_Unprovable;
```

```
package body Counterex_Unprovable with
  SPARK_Mode
is

   function Double (X : Int) return Int is
   begin
      return 2 * X;
   end Double;

   procedure Double_In_Call (X : in out Int) is
   begin
      X := Double (X);
   end Double_In_Call;

   procedure Double_In_Loop (X : in out Int) is
      Result : Int := 0;
   begin
      for J in 1 .. 2 loop
         Result := Result + X;
      end loop;
      X := Result;
   end Double_In_Loop;

end Counterex_Unprovable;
```

両方の場合において， GNATprove が生成する反例は，検証器が誤って， X 入力値が 1 のとき， -3 を出力すると演繹するからです．呼び出される関数に契約が欠落している或いはループ実行時のループ不変条件が不足しているのが原因です．

```
counterex_unprovable.adb:7:16: info: overflow check proved
counterex_unprovable.adb:7:16: info: range check proved
counterex_unprovable.adb:12:12: info: precondition proved
counterex_unprovable.adb:19:20: info: initialization of "Result" proved
counterex_unprovable.adb:19:27: medium: range check might fail (e.g. when Result␣
↪= -100 and X = -1)
counterex_unprovable.adb:21:12: info: initialization of "Result" proved
counterex_unprovable.ads:8:14: info: overflow check proved
counterex_unprovable.ads:9:14: info: overflow check proved
counterex_unprovable.ads:9:14: info: postcondition proved
counterex_unprovable.ads:12:14: info: overflow check proved
counterex_unprovable.ads:13:14: medium: postcondition might fail, cannot prove X␣
↪= 2 * X'old (e.g. when X = -3 and X'Old = -1)
counterex_unprovable.ads:13:20: info: overflow check proved
counterex_unprovable.ads:16:14: info: overflow check proved
counterex_unprovable.ads:17:14: medium: postcondition might fail, cannot prove X␣
↪= 2 * X'old (e.g. when X = -3 and X'Old = -1)
counterex_unprovable.ads:17:20: info: overflow check proved
```

2. 検証器の能力不足により（詳細は次の節を参照下さい **証明器の不足要素を調査する** ），プロパティが証明できないとき，反例を見ることで，検証器がそのプロパティを証明できなかった理由を知ることができる場合があります．しかし，どうして証明機がプロパティを証明できなかったを説明するに過ぎません．ちなみに，CVC4 検証器を使った時のみ常に反例が生成されます．また，もし CVC4 検証器が選択されなれなくても， `--no-counterexample` スイッチによって反例の生成が無効になっていないならば，CVC4 を用いて反例を生成しようと試みます．しかし，CVC4 の証明結果は，この場合含まれません．
3. タイムアウトないしはステップ数に小さな値を用いたとき，検証器は，完全な反例を生成する前に，資源制約に掛かるかもしれません．その場合，生成された反例は，適切な実行に対応していないかもしれません．

4. `--proof` スイッチの値が，デフォルト値である `per_check` であり，サブプログラムの全てのパスにおいて，反例は変数の値を出力します．適切な実行に対応しているパスだけではありません． `progressive` 或いは `per_path` を用いて， GNATprove を再実行することで，反例における可能な実行パスを分離することができます．

6.3 GNATprove をチームで使う方法

最も一般的な GNATprove 利用法は，チーム内での通常の品質管理ないしは品質保証アクティビティの一部としての利用です．通常， GNATprove を，現在のコードベースに対して毎晩実行します．開発者は，自分のコンピュータないしはサーバー上で昼間に GNATprove を実行します．この夜間と昼間の実行で得られた GNATprove の結果は，チームメンバで共有する必要があります．結果を見たり，新しい結果を共有している他の結果と比較します． GNATprove を実行し，結果を共有することで，様々なプロセスで利用することができます．

全てのケースで，ソースコードを（例えば共有ドライブ上で）直接に共有すべきではありません．この場合，ファイルのアクセス権や同時アクセスの問題を引き起こすことになります．従って，典型的な利用法としては，各ユーザがソースコードや環境をチェックアウトすることです．物理的に全てのユーザでソースコードを共有するのではなく，自分たちのソースコードのバージョン／コピーやプロジェクトファイルを使用します．

また，プロジェクトファイル中では常に，ローカル・非共有・書き込み可能ディレクトリをオブジェクトディレクトリとして指定します（明示的か暗黙的かに関わらず，オブジェクトディレクトリが存在しないと，プロジェクトファイルが置かれているディレクトリがオブジェクトディレクトリとして利用されます）．

6.3.1 幾つかのワークフロー

GNATprove をチームで利用するための幾つかのワークフローがあります．

1. GNATprove を，サーバないしはローカルで実行します．警告や検査メッセージは出力されないようにします．誤った警告を抑制し，証明されていない検査メッセージを正当化することで，可能となります．
2. GNATprove を，サーバないしはローカルで実行します．テキスト形式の結果を，構成管理で共有します
3. GNATprove を，サーバで実行します．結果を示すテキストを，サードバーティの品質計測ツール（例えば，GNATdashboard, SonarQube, SQUORE など）に送ります
4. GNATprove を，サーバないしはローカルに実行します．その上で，GNATprove セッションファイルを，構成管理で共有します．

全てのワークフロー（最初のワークフローでは必須となりますが）において，メッセージを抑制し正当化することができます．全ての正当で完全な検証ツールと同様に，もちろん GNATprove が誤った警告をだすかもしれません．メッセージのタイプを見つけることが最初に必要です．

- 警告を抑制することができます．次を見て下さい． **警告を抑制する**
- 検査メッセージを正当化することができます．次を見て下さい． **検査メッセージを正当化する**

証明から得ることの出来る検査メッセージは，検証可能な検査とも関係しています．このためには，正しい契約 かつ／または 解析スイッチを見つけるために GNATprove とやりとりする必要があります．詳しくは次を見て下さい． **証明されなかった検査項目を調査する方法**.

ワークフロー 3 におけるテキスト出力は，次のものと関係しています．GNATprove が出力するコンパイラのような出力で， `--report` スイッチと `--warnings` を用います（詳細については， **コマンドラインで *GNATprove* を使う** を参照下さい）．デフォルトでは，メッセージは証明に失敗した検査と警告に対してのみ発行されます．

ワークフロー 2 におけるテキスト出力は，上記のコンパイラのような出力と， GNATprove が生成するファイル `gnatprove.out` (詳細については次を参照して下さい **出力におけるモードの影響** 及び **仮定管理**) 中の付加的情報です．

ワークフロー 4 は， 各ソースパッケージに対する形式検証の状態を記録するために， GNATprove が用いているセッションファイルを共有する必要があります．これは， Project Attributes 中に `Proof_Dir` 証明ディレクトリを記述し，このディレクトリを構成管理下で共有することにより得られます．衝突を避けるために，開発者は，このディレクトリにおける局所的な変更を構成管理に入れないことを推奨します．その代わり，定期的にディレクトリの更新されたバージョンを入出するようにします．例えば，サーバー上で夜間に実行するか，担当のチームメンバーが， GNATprove によって生成された最新のバージョンとともに証明ディレクトリを更新する責任者とすることができます．

他のワークフローと比較したときのワークフロー 4 のメリットは，すでに証明されているプロパティをローカルに再証明することを避けるという点にあります．共有セッションファイルは，証明された検証状態（Verification Conditions）の痕跡を保ち続けることができるからです．

6.3.2 警告を抑制する

GNATprove は，2 種類の警告を発行し，別々に制御することができます．

- コンパイラの警告は，通常の GNAT コンパイラスイッチによって制御できます．
  - `-gnatws` 全ての警告を抑制する
  - `-gnatwa` 全ての付加的な警告を可能とする
  - `-gnatw?` 最後の文字によって示される特定の警告を可能とする

    詳細については GNAT ユーザガイドを参照下さい．プロジェクトファイル中に記述されたコンパイルスイッチによって，コンパイラに渡すこともできます．
- GNATprove 特有の警告は， `--warnings` スイッチによって制御することができます．
  - `--warnings=off` 全ての警告を抑制する
  - `--warnings=error` 警告をエラーとして扱う
  - `--warnings=continue` 警告を発行するが，解析は停止しない（デフォルト）

    デフォルトでは GNATprove は，警告を出しますが，解析は停止しません．

ソースコード中で， `Warnings` pragma を用いることで，どちらのタイプの警告も選択的に抑制することができます．例えば， GNATprove は，手続き `Warn` 中で，3 つの警告を発行するとします．これらはソースコード中で，3 つの `Warnings` pragma を使用することで抑制できます．

```
1  pragma Warnings (Off, "unused initial value of ""X""",
2                   Reason => "Parameter mode is mandated by API");
3
4  procedure Warn (X : in out Integer) with
5    SPARK_Mode
6  is
7     pragma Warnings (Off, "initialization has no effect",
8                      Reason => "Coding standard requires initialization");
9     Y : Integer := 0;
10    pragma Warnings (On, "initialization has no effect");
11
12 begin
13    pragma Warnings (Off, "unused assignment",
14                     Reason => "Test program requires double assignment");
15    X := Y;
16    pragma Warnings (On, "unused assignment");
17    X := Y;
18 end Warn;
```

ここで Warnings Off pragma から開始し， Warnings On pragma ないしは関係するスコープの終了場所で囲まれた領域内で，特定のメッセージを持った警告を，抑制することができます． Reason 根拠文字列を使用することもできます．ある特定の形式を持った全ての警告を抑制するために，正規表現を用いて，特定のメッセージの代わりに与えることができます．プロジェクトの全てのユニットで横断的に，関連する警告を抑制するために，構成ファイル中に， Warnings Off pragma を加えることができます．特定のエンティティに Warnings Off pragma を指定することで，このエンティティに関係した全ての警告を抑制することができます．

この Warnings pragma は，GNAT コンパイラ或いは GNATprove に対してのみ適用することを示すために GNAT 或いは GNATprove を第一引数にとることも可能です．

```
pragma Warnings (GNATprove, Off, "unused initial value of ""X""",
                 Reason => "Parameter mode is mandated by API");

procedure Warn2 (X : in out Integer) with
  SPARK_Mode
is
   pragma Warnings (GNATprove, Off, "initialization has no effect",
                    Reason => "Coding standard requires initialization");
   Y : Integer := 0;
   pragma Warnings (GNATprove, On, "initialization has no effect");

begin
   pragma Warnings (GNATprove, Off, "unused assignment",
                    Reason => "Test program requires double assignment");
   X := Y;
   pragma Warnings (GNATprove, On, "unused assignment");
   X := Y;
end Warn2;
```

洗練したバージョンである Warnings pragma を用いることの利点は文書化という他に，スイッチ -gnatw.w とともに用いることで，警告を全く抑制しない意味の無い Warnings pragma を見つけることができます．実際このスイッチは，コンパイル中は GNAT とともに使うことができ，形式検証中は，GNAT prove とともに使うことができます．これは， Warnings pragma をどちらかのツールのみに適用できることと同様です．

詳しくは， GNAT 参照マニュアルをご覧下さい．

6.3.3 情報メッセージを抑制する

情報メッセージは，ソースコード中で，pragma Warnings を用いることで，警告と同様に，抑制することができます．

6.3.4 検査メッセージを正当化する

Annotate Pragma を用いた直接的正当化

GNATprove のフロー解析或いは証明により生成した検査メッセージは，ソースコード中に Annotate pragma を追加することで，選択的に正当化できるようになります．例えば，return 文における潜在的なゼロ割についての検査メッセージは，次のように正当化することができます．

```
return (X + Y) / (X - Y);
pragma Annotate (GNATprove, False_Positive,
                 "divide by zero", "reviewed by John Smith");
```

pragma は次の形式を持ちます．

```
pragma Annotate (GNATprove, Category, Pattern, Reason);
```

ここで，次のテーブル表現は，異なったエントリーについての説明です.

項目	説明
GNATprove	固定の識別子である
Category	`False_Positive` か `Intentional` のいずれかである
Pattern	正当化されるべき検査メッセージのパターンを記述した文字列リテラルである
Reason	レビューのための正当化を提供する文字列リテラルである

全ての論拠を示す必要があります.

項目 *Category* は現時点では，ツールのふるまいになんの影響も与えません．文書化の目的のみです.

- `False_Positive` は， GNATprove は証明できていませんが，検査が不合格ではないということを示します.
- `Intentional` 検査は不合格ですが，バグではないと考えていることを示しています.

Pattern は，正当化すべき検査メッセージの副文字列になります.

Reason は，正当化としてレビューに対してユーザが示す文字列です．この論拠（reason）は， GNATprove のレポート中に記載されます.

記載方法に関するルールは次の通りです．命令文リストないしは宣言文リスト中で， `Annotate` pragma は，リスト中の先行するアイテムに適用され，他の `Annotate` pragma を無視します．もし，先行するアイテムがない場合は，pragma は，内包構成要素に適用されます．例えば，if 文で，ガード条件が成立したときの処理（then-branch）における最初の要素がこの pragma であれば，if 文中のガード条件に適用されます.

もし先行するあるいは内包構成要素が，サブプログラムのボディ部である場合，pragma は，サブプログラムのボディ部と契約を含んでいる仕様部に適用されます．このことで，呼び出し元に関係しているとき，仕様部で GNATprove が出力する検査メッセージに対する正当化を行うことができます.

注意点としては，ソースコード中で，pragma `Annotate` の書かれたあと，極力広い範囲にわたって， pragma が適用されるということです.

- （新しい）pragma が，ブロック中の命令文リストに現れたときは，それがブロック全体（if 文中では，全ての分岐を含む if ブロック内，ループ文の場合は，ループ文全体）に適用されることになります.
- pragma が，サブプログラムボディ部の先頭にあるとき，その pragma は，ボディ部全体とサブプログラムの仕様部に適用されます.

ユーザは，適切に， `Annotate` を記載すべきです．そうすべきではない検査を正当化しないように注意する必要があります.

```
procedure Do_Something_1 (X, Y : in out Integer) with
  Depends => ((X, Y) => (X, Y));
pragma Annotate (GNATprove, Intentional, "incorrect dependency ""Y => X""",
                 "Dependency is kept for compatibility reasons");
```

或いは，そのユニットのユーザにとって不可視である必要があるという実装時選択があるときは，ボディ部上で正当化を行います.

```
procedure Do_Something_2 (X, Y : in out Integer) with
  Depends => ((X, Y) => (X, Y));
```

```
procedure Do_Something_2 (X, Y : in out Integer) is
begin
   X := X + Y;
```

```
      Y := Y + 1;
   end Do_Something_2;
   pragma Annotate (GNATprove, Intentional, "incorrect dependency ""Y => X""",
                    "Currently Y does not depend on X, but may change later");
```

検査メッセージに対する正当化を行わない上記のような Annotate pragma 形式は，役に立たず， GNATprove は警告を発行します． GNATprove が発行する他の警告と同様に，この警告は， --warnings=error スイッチが設定された場合，エラーとして扱われます．

Pragma Assume による間接的正当化

GNATprove の証明によって生成される検査メッセージは，ソースコード中に， *Pragma Assume* を付加することによって，代替として間接的に正当化することができます．これによって，検査項目は証明されます．例えば，以下に示す割り当て文において整数オーバーフローの可能性に関する検査メッセージは，次のように正当化することができます．

```
   procedure Next_Tick is
   begin
      pragma Assume (Clock_Ticks < Natural'Last,
                     "Device uptime is short enough that Clock_Ticks is less than 1_
↪000 always");
      Clock_Ticks := Clock_Ticks + 1;
   end Next_Tick;
```

pragma Assume は， Annotate pragma を利用するよりも強力です．このプロパティの仮定は，一つ以上の検査項目の証明に用いることができるからです．従って，単純な実行時検査を正当化するためには， Assume pragma を用いるよりも，一般的には， Annotate pragma を用いるべきです．以下の幾つかの例で Assume を使用することが望ましい場合を示します．

- 仮定を局所化するために：

  ```
  pragma Assume (<External_Call's precondition>,
                 "because for these internal reasons I know it holds");
  External_Call;
  ```

  もし， External_Call の事前条件が変化した場合，仮定は引き続き論拠に対して有効であっても，ここでの仮定は有効ではなくなるかもしれません． External_Call の事前条件中の変化によって，ここで不整合が生じると， External_Call の証明ができなくなります．

- レビューを容易化するために，対象の外部から期待されていることをまとめておく：

  ```
  External_Find (A, E, X);
  pragma Assume (X = 0 or (X in A'Range and A (X) = E),
                 "because of the documentation of External_Find");
  ```

  幾つかの progma Annotate を用いて情報を分散するよりも，たった一つの pragma Assume を用いる方が，保守やレビューが容易になります．もし，複数の場所で情報が必要ならば，pragma Assume を幾つかの手続きに分割します．

  ```
  function External_Find_Assumption (A : Array, E : Element, X : Index) return␣
   ↪Boolean
  is (X = 0 or (X in A'Range and A (X) = E))
  with Ghost;

  procedure Assume_External_Find_Assumption (A : Array, E : Element, X : Index) with
   Ghost,
  ```

```
 Post => External_Find_Assumption (A, E, X)
is
   pragma Assume (External_Find_Assumption (A, E, X),
                  "because of the documentation of External_Find");
end Assume_External_Find_Assumption;

External_Find (A, E, X);
Assume_External_Find_Assumption (A, E, X);
```

一般的に，仮定は可能な限り小さくすべきです（コードを動作させるための必要なもののみとする）．pragma `Assume` を用いた間接的正当化は，注意深く調べるべきです．なぜならば，検証プロセス中に容易にエラーを持ち込む可能性があるからです．

6.3.5 仮定管理

GNATprove は，サブプログラムとパッケージを別々に解析するので，その結果は，解析されないサブプログラムとパッケージに関する仮定に依存します．例えば，サブプログラムには実行時エラーがないという検証は，プロパティに依存します．このプロパティとは，呼び出している全てのサブプログラムにおいて実装済みの契約です．もし，あるプログラムが， GNATprove によって完全に解析されるならば，仮定の相互参照は，ほぼ自動的に行われます（いくつかの例外はあります．呼び出しの無限連鎖のために検査ができない場合です）．しかし，一般に，プログラムの一部は SPARK で記述され，他の部分は他の言語，多くは Ada, C, アッセンブリで記述されます．このように， GNATprove で解析することができない部分についての仮定は，他の手段（テスト・手動による解析・レビュー）で検証するために記録しておく必要があります．

スイッチ `--assumptions` が設定されたとき, GNATprove は，その結果ファイル `gnatprove.out` 中の残っている仮定についての情報を出力します．残っている仮定は，望む検証目的に適合するよう正当化するために必要です．サブプログラムについての仮定が生成されますが，様々なケースがあります．

- サブプログラムが解析されなかった（例えば， `SPARK_Mode => Off` と指定されていた）
- サブプログラムは， GNATprove で完全には解析されなかった（即ち，幾つかの検証されていない検査項目が残っている）

現在は，呼ばれるサブプログラムにおける仮定のみが出力され，呼び出しているサブプログラムの仮定が出力されないことに注意して下さい．

以下のテーブルは， GNATprove が出力するかもしれない仮定と，その主張の意味を説明しています．

仮定	説明
effects on parameters and global variables	サブプログラムは，仕様部（シグネチャ＋データ依存）に記述されているパラメータないしは広域変数以外の読み書きをしていない．
absence of run-time errors	サブプログラムには実行時エラーがない
the postcondition	サブプログラムの事後条件は，サブプログラムの各呼び出し後も保持されている

6.4 証明されなかった検査項目を調査する方法

形式検証におけるもっとも挑戦的なアスペクトの一つは，失敗した証明を解析することです． GNATprove が，実行時検査や表明の保持に関して，自動的な証明で不合格としたとき，そこには幾つかの理由があります．

- [CODE] コードが間違っているため，検査項目ないしは表明が保持されていない
- [ASSERT] 表明が間違っており，正しく保持されていない
- [SPEC] プログラムのふるまいについて幾つかの不足する表明があり，表明の検査が証明できない

- [MODEL] 現在 GNATprove で用いているモデルには限界があり，検査や表明が証明できない
- [TIMEOUT] 検証器のタイムアウトのため，検査或いは表明が証明できない
- [PROVER] 検証器が不十分で，検査或いは表明が証明できない

6.4.1 間違っているコードあるいは表明を調査する

最初のステップは，コードが誤っている [CODE] 或いは，表明が誤っている [ASSERT] か，或いはその双方かを検査することです．実行時検査や表明は，実行時に行われるので，コードや表明の正しさに関する信頼を増加させる一つの方法は，代表的な入力を用いてプログラムをテストすることです．次の GNAT スイッチを用いることができます．

- `-gnato`: 演算途中のオーバーフローの実行時検査を可能とする
- `-gnat-p`: 全ての検査を抑制するために `-gnatp` が使われていても，実行時検査を再度可能とする
- `-gnata`: 表明の実行時検査を可能とする

6.4.2 証明できないプロパティを調査する

二番目のステップは，プロパティが証明可能であるかどうかを考えることです [SPEC]. 検査ないしは表明は，必要な注記 (annotation) が欠けているために，証明不能であるかもしれません．

- 内包しているサブプログラムの事前条件が弱すぎる，或いは
- 呼び出されるサブプログラムの事後条件が弱すぎる，或いは
- 内包しているループに対するループ不変条件が弱すぎる，或いは
- 検査や表明の前のループに対するループ不変条件が弱すぎる

特に， GNATprove は，サブプログラムのボディ部を調べないので，呼び出しについて必要な全ての情報は，サブプログラム契約中で明確であるべきです．コードと表明に焦点を当てた手動レビューによって，効率的に，多くの失われた注釈について判断することができます．表明の記述が多くても， GNATprove は，正確に自分が証明できない場所を示すことができます．こうすることで，問題を明らかにする助けとなります．調査しつつ，コードを単純化することは役立ちます．例えば，単純な表明を追加して，証明してみることです．

GNATprove が提供するパス情報は，コードレビューで役に立ちます．証明が失敗したパスを，*GPS で GNATprove を実行する* で記述したように，エディタ内で表示することができます．場合によっては，反例もまたパス上に生成されます．このとき，問題を示す変数の値も分かります（詳しくは， **反例を理解する** 参照のこと）．多くの場合，不足している表明を見つけ出すには，これで十分です．

プロパティもまた，概念的には証明可能です．しかし， GNATprove が用いるモデルでは，今のところモデルを用いた推論を行うことができません [MODEL] ．特に，下記の言語の特徴を利用すると，真になるべき VC（Verification Condition，検証条件） を生成できる場合があります．ただし，証明することはできません．

- 浮動小数点算術演算（しかし， CodePeer integration，役立つ）
- 文字列リテラルの内容（こちらも， CodePeer integration を用いることで支援可能である）

CodePeer integration を使用するためには, スイッチ `--codepeer=on` を GNATprove に渡します. CodePeer も他の検証器が存在しないケースにおいては，不足情報を `pragma Assume` を用いて補うことで，VC を証明することができます．

証明不能なプロパティと証明器に不足する要素がある（次章）ことを区別するのは，時には困難です．この問題を中心的な問題に限定するために，一般的な最も役に立つアクションは，コード中に表明を挿入することです．この表明では，プログラム中のある特定の点で，そのプロパティ（或いはその一部）が証明可能かどうかを **テスト** します．例えば，もし，ある事前条件で次の様なプロパティを設定しているとします： (P or Q) いま，

実装が多くの分岐とパスを含んでいるとします．ことの時に次の表明を加えることを考えます．適切な場所において，P が真であるか，或いは Q が真であるか．これは，次の 2 つのケースを区別するために役立ちます．

- 証明不可能なプロパティの場合，プログラム中の特定のパスが示され，問題を引き起こす特定のプロパティの箇所が分かる可能性があります．
- 証明器に不足が有る場合，こうすることで，証明器が，表明とプロパティの双方をなんとか証明する助けになります．自動証明に役立つ表明のみをコード中に残し，証明器とやりとりする中で挿入する他の表明を取り除くというのは，良い方法です．

6.4.3 証明器の不足要素を調査する

最後のステップは，証明器が証明に十分な時間を与えられたかどうか [TIMEOUT] または，別の証明器が証明を見つけられるかどうか [PROVER] です．この目的のために， GNATprove は， `--level` スイッチを提供し，コマンドラインから（**コマンドラインで** *GNATprove* **を使う** 参照）或いは，GPS から（*GPS* **で** *GNATprove* **を実行する** ）或いは GNATbench から（*GNATbench* **から** *GNATprove* **を実行する** 参照）から利用することができます．デフォルトの 0 レベルは，単純な証明に十分です．一般的に，自動証明がそれ以上得られなくなるまで，証明のレベルを（レベル 4 まで）あげることができます．

次の節 **コマンドラインで** *GNATprove* **を使う** で記述するように，スイッチ `--level` は，さまざまなより低いレベルのスイッチ `--steps` , `--prover` , `--proof` を直接設定するのと等価です．それゆえ， `--level` における既定義の組み合わせを用いるよりも，より強力な（必然的により長い証明時間を必要とする）値を設定することができます．

上記の実験に関して，GPS の *SPARK* → *Prove Line* 或いは，*SPARK* → *Prove Subprogram* メニューを使うと便利です． *GPS* **で** *GNATprove* **を実行する** および *GNATbench* **から** *GNATprove* **を実行する** に記載があります．対象となる行ないしはサブプログラムに対して，より高速に結果を得ることができます．

現在の自動証明器には，浮動小数点算術演算をきわめて正確には扱えないという問題があります．特に，多数の演算がある，或いは非線形演算（乗算・除算・累乗）を含んでいる場合です．この場合は， CodePeer integration を用いるのが便利です．スイッチ `--codepeer=on` で動作し，浮動小数点演算の境界検査を高速かつ正確に実施することができます．

自動証明に共通に存在する限界は，非線形の算術を適切に扱えないことです．例えば，乗算，除算，モジュロー演算，累乗を含んでいる単純な検査の証明に失敗する場合があります．

その場合，ユーザは次のいずれかの方法をとることができます．

- コード中に，SPARK 補題ライブラリからの補題の呼び出しを加えます．
- ユーザの補題呼び出しをコード中で行う．
- コード中に仮定を付け加える．
- コード中に正当化を付け加える．
- 証明不可の検査を手動でレビューし，その結果を信頼できる形で，記載します．(例えば，版管理下で，GNATprove の結果を保存する）

将来的には， GNATprove は，式の *user view* を提供し，証明器に渡す予定です．これは，上級者が調査を行うためです．このビューは，Ada 風の構文を持ち，証明に失敗した実際の式を表現します．これによって，ユーザは，その解釈が容易になります．現在，この形式はまだ定義されていません．

特に手動で証明したい上級者のために, GNATprove が生成する証明ファイルの形式を提供する予定です．これによってユーザは，証明器に渡される実際のファイルを理解することができるようになります．個々のファイルは，プロジェクトオブジェクトディレクトリ（デフォルトはプロジェクトディレクトリ）の `gnatprove` サブディレクトリ下に保持されます．ファイル名は以下の名前付けとなります:

```
<file>_<line>_<column>_<check>_<num>.<ext>
```

ここで:

- `file` は，検査対象の Ada ソースファイルの名前です
- `line` は，検査する行です
- `column` は，カラムです
- `check` は，検査の識別子です
- `num` は，補足の数字で，プログラムにおける異なるパスの識別です．パスは，サブプログラムの開始と検査位置の間にあります．
- `ext` は，選択したファイル形式に関係する拡張です．これは，証明器に依存します．例えば， Alt-Ergo の場合は， Why3 形式となります．CVC4 に対するファイルは，SMTLIB2 形式となります．

例えば，Alt-Ergo 証明器で， `f.adb` ファイルの 160 行目，42 カラム目の範囲検査に対して生成される証明ファイルは，以下に保持されます:

```
f.adb_160_42_range_check.why
f.adb_160_42_range_check_2.why
f.adb_160_42_range_check_3.why
...
```

証明器 CVC4 に対して生成する同様の証明ファイルは:

```
f.adb_160_42_range_check.smt2
f.adb_160_42_range_check_2.smt2
f.adb_160_42_range_check_3.smt2
...
```

これらのファイルを調査するためには， GNATprove に対して，コマンドラインで `-d` スイッチを付け加えることで，これらファイルを保持するように指示することができます．また， `-v` を用いることで， GNATprove が生成し，証明しようと試みる証明ファイルの詳細なログを入手できます．