SPARK Tutorial - Schreiben sicherer Software

Felix Trojan

April 2010

Abstract

Bestimmte Einsatzgebiete für Software stellen besondere Anforderungen an Sicherheit und Stabilität von Programmen. Überall dort wo ein Programmierfehler fatale Auswirkungen hätte zB. Menschenleben oder gravierende finanzielle Folgen muss mit besonderer Sorgfalt gearbeitet werden. SPARK bietet hierfür eine auf Ada basierende Programmiersprache mit der Programme entwickelt werden können die "nahezu" Bugfrei sind und mit speziellen Programmen der SPARK Toolsuite mathematisch bewiesen werden können.

Contents

1 Voraussetzungen

Bevor man dieses Tutorial beginnt sollte man folgende Voraussetzungen mitbringen:

- allgemein fit im Programmieren sein
- relativ gute Ada-Kenntnisse
- relativ gutes Verständnis für Mathematik(Logik, Ungleichungen)

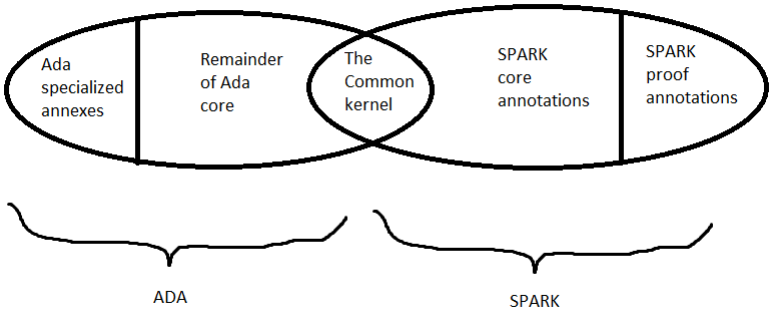
2 Was ist SPARK?

SPARK ist eine Programmiersprache für die Entwicklung sicherer und stabiler Software diese findet vorallem Anwendung in:

- Luftfahrtelektronik
- Medizinische Systeme
- Prozesssteuerung zB. in Kraftwerken
- Finanzsoftware zB. Banking, Versicherungen

SPARK ist eine Untermenge von Ada, das bedeutet dass alle programmkritischen Merkmale wie zB. dynamische Speicherallokation aus Ada entfernt wurden, da diese die häufigsten Programmierfehler beherbergen. Da SPARK-Programme in Ada geschrieben sind werden sie auch mit einem Ada-Kompiler(zB. GNAT) kompiliert. Das besondere ist aber das SPARK-Programme mit speziellen Ada-Kommentaren versehen werden müssen, die sog. Annotations. Diese ermöglichen es den Programmcode SPARK-konform zu gestalten und ihn später mit den SPARK-Analysetools zu beweisen. Diese Beweisführung ist es die ein SPARK-Programm ausmacht, denn es kann immer, bei korrektem Beweis, sichergestellt werden dass das Programm frei von Bugs ist. Später werden wir sehen dass eine völlige Bugfreiheit nicht immer zu gewährleisten ist, vorallem wenn es um Zugriffe auf Hardware o.Ä. geht.

3 Was ist in SPARK verboten?



Einschränkungen SPARKs gegenüber Ada

- keine Exceptions, Generics(Templates)
- keine Access(Pointer) Types, Goto Statements
- kein "Ausbrechen" aus Schleifen
- keine dynamische Speicherallokation
- keine dynamischen Arrays
- keine Überladung von Unterprogrammen
- keine Variablenverdeckung im Scope
- Subtypen müssen explizit benannt werden
- Alle Constrains(Einschränkungen) sind statisch
- Keine Rekursion (ist jedoch möglich wenn man zB. eine Aufruftiefe festlegt)
- Kein ungeprüftes Casting zwischen nichtverwandten Typen
- keine doppelten Namen
- Paket-Unterprogramme und -Variablen müssen mit vollem Pfad angesprochen werden(kein use)

SPARK mag einen anfangs sehr kleinlich und eingeschränkt vorkommen, dennoch haben diese Einschränkungen alle ihre Berechtigung. Bei der Entwicklung von SPARK hat man sich verschiedene

Schutzziele ausgedacht aus denen die obigen Einschränkungen hervorgehen. Es müssen starke Annahmen über Bezeichner getroffen werden, deshalb bekommt **alles** explizit einen eigenen Namen(Subtypen, Variablen, keine Überladung). Weitere sind dass die geschriebenen Programme endlich in Speicherbelegung und Abarbeitung ihrer Algorithmen sind(keine dynamische Speicherallokation). Programme müssen einen Eingang und eine Endliche Zahl von Ausgängen besitzen, es darf deshalb auch nicht im Code herumgesprungen werden.

4 Schreiben eines SPARK-Programmes

Wir brauchen vorerst:

- Texteditor oder IDE nach Wahl
- irgendeinen Ada-Compiler zB. GNAT welcher auch im GCC enthalten ist

Eine kostenlose Distribution der SPARK-Tool-Suite(Windows,Linux) findet man auf:

`http://libre.adacore.com/libre/download/`

Diese brauchen wir aber erst später zunächst befassen wir uns mit ein paar Grundlagen.

4.1 Programmaufbau

4.1.1 Packete

SPARK unterscheidet eine strikte Trennung von Paketen in:

Specification Files *.ads(Header) und **Body Files *.adb (Implementation)**

Beispiel 1

foo.ads

```
package foo
is
procedure foo1;
 function  foo2(X : Integer) return Integer;
end foo;
```

foo.adb

```
package body foo
is
 procedure foo1 is
 begin
  -- some code
 end foo1;
 function  foo2(X : Integer) return Integer is
 begin
  -- some code
  return --someting;
 end foo2;
end foo;
```

Beispiel 2 - Modelierung eines Leergutautomaten
Annotations zur vereinfachung erstmal ausgelassen.

leergut.ads

```
with Spark_IO;

package Leergut
   is
   subtype Money is Float range 0.0 .. 400.0;

   procedure print_sum;

   procedure add_008;

   procedure add_015;

   procedure add_025;

   procedure reset_sum;

private
  Psumme: Money := 0.0;
end Leergut;
```

leergut.adb

```
package body Leergut is

   procedure print_sum is
   begin
     Spark_IO.Put_Line(Spark_IO.Standard_Output,"Summe:",6);
     Spark_IO.Put_Float(Spark_IO.Standard_Output,Psumme,3,2,2);
     Spark_IO.New_Line(Spark_IO.Standard_Output,1);
   end print_sum;

   procedure add_008 is
       tmp : Money;
   begin
      tmp := Psumme + 0.08;
      if (tmp > 300.0)
      then
            print_sum;
            Psumme := tmp - Psumme;
      end if;
     Psumme := Psumme + 0.08;
   end add_008;

   procedure add_015 is
   tmp : Money;
   begin
      tmp := Psumme + 0.15;
      if (tmp > 300.0)
      then
            print_sum;
            Psumme := tmp - Psumme;
      end if;
     Psumme := Psumme + 0.15;
   end add_015;

   procedure add_025 is
    tmp : Money;
   begin
      tmp := Psumme + 0.25;
      if (tmp > 300.0)
      then
            print_sum;
            Psumme := tmp - Psumme;
      end if;
     Psumme := Psumme + 0.25;
   end add_025;

   procedure reset_sum is
   begin
      Psumme := 0.0;
   end reset_sum;

end leergut;
```

Desweiteren ist zu beachten das sogenannten Nesting, d.h. definieren von Paketen innerhalb anderer Pakete, verboten ist.

foo.ads

```
package foo
is
 procedure foo1;
 function  foo2(X : Integer) return Integer;

 package foo2
 is
  -- verboten!
 end foo2;

end foo;
```

4.2 Hauptprogramm

Die "Main" eines SPARK-Programmes besteht aus einer *.adb mit nur einer Prozedur welche den gleichen Namen wie der Datei-Präfix besitzt. Die Procedure ist dabei "standalone", darf sich also in keinem Packet befinden. Desweiteren wird es durch eine spezielle Annotation −−# main_program; gekennzeichnet.

sparkmain.adb

```
--# main_program;
procedure sparkmain
is
   a:Integer; --lokale variable
begin
  --some code
end sparkmain;
```

4.3 Annotations

Eine Annotation (dt. Vermerk, Anmerkung) ist ein spezieller Ada-Kommentar der Form:

−−# "Reserviertes Wort"

Durch die Anreicherung des Codes mit Annotations bekommt dieser zum einen mehr Ausdruckskraft, zum anderen kann er nun von der SPARK-Tool-Suite analysiert werden. Der SPARK Examiner, eines der SPARK-Tools, prüft dabei auf Konformität zwischen Code und Annontations und kann anhand dieser den Informationsfluss analysieren, Run Time Checks einfügen und mathematische Hypothesen und Folgerungen(Konklusionen) aufstellen,die sog. Verification Conditions ,welche es später zu beweisen gilt.

Übersicht über die Motivationen von Annotations

- Kontrolle über Sichtbarkeiten von Paketnamen, Variablen
- Kennzeichnen des Informationsfluss(Import/Export,Pfade)
- Treffen von Annahmen(Assertions)
- Stellen von Pre- und Post-Conditions
- Auskunft über Inhalte von Paketen und Initialisierung von Variablen
- Verstecken von programmkritischem Code

4.3.1 Subprogram Annotations

Aufgabe der Subprogram Annotations ist es den Infromationsfluss unserer Unterprogramme aufzuzeigen. Dabei unterscheidet man zusätzlich in Annotations für procedures(global, derives) und für functions(return).

Global Annotation

Die Global Annotation gibt die vom Unterprogramm verwendeten globalen Variablen an. Dabei wird angeben ob diese importiert(IN) oder exportiert(OUT) wird, ähnlich wie bei den Parametern. Diese Annotation ist Pflicht sobald eine globale Variable verwendet wird, ansonsten entfällt sie.

```
Syntax:
--# global [mode] variableName , variableName;
Examples:

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

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

Beispiel Leergut 4.1.1 für `procedure add_008`

```
.....
procedure add_008;
--#global in out Psumme;
.....
```

Im Body des Pakets greifen wir sowohl lesend als auch schreibend auf Psumme zu:

```
.....
 procedure add_008 is
       tmp : Money;
   begin
      tmp := Psumme + 0.08;
      if (tmp > 300.0)
      then
            print_sum;
            Psumme := tmp - Psumme;
      end if;
     Psumme := Psumme + 0.08;
   end add_008;
.....
```

Derives Annotation

Diese Annotation ist Pflicht für alle procedures und kennzeichnet den Informationsfluss zwischen Parametern untereinander und eventuellen globalen Variablen.

```
Syntax:
--# derives [dependencyClause&dependencyClause];
Examples:
procedure Flt_Integrate(Fault : in Boolean;
Trip : in out Boolean;
Counter : in out Integer)
--# derives Trip from *, Fault, Counter &--# Counter from *, Fault;

procedure BusyWait
--# derives ;
```

Beispiel Leergut 4.1.1 für `procedure add_008`

```
  Psumme := tmp - Psumme;
```

Hier wird eine neue Psumme aus der alten Psumme berechnet. Darum fügen wir folgende Annotation hinzu:

```
.....
procedure add_008;
--#global in out Psumme;
--#derives Psumme from *;
.....
```

Das * ist eine kurzschreibweise für `--#derives Psumme from Psumme`.
Aber Achtung!

```
--# derives A,B,C from *,X,Y;
```

`kann man auch schreiben als`

```
--# derives A from A,X,Y &
--#         B from B,X,Y &
--#         C from C,X,Y;
```

Desweiteren gibt es **Null Derives**, diese verwendet man wenn importierte Variablen keine sichtbaren Exporte aufweisen. Null Derives müssen den anderen Derives nachgestellt, also als letztes aufgelistet werden. Siehe auch 5.3

`Example:`

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

Return Annotation

Zuerst einmal muss man sagen das Functions in SPARK eine gesonderte Rolle spielen. Sie sind nämlich als Funktionen im mathematischem Sinn zu verstehen, d.h. sie bekommen Eingabeparamter und liefern eine dazugehörige Ausgabe, **aber** sie nehmen im Gegensatz zu procedure keinen Einfluss auf Parameter oder globale Variablen, sie verändern also den "State" des Programmes nicht, sondern haben nur eine beobachtende Funktion. Deshalb gibt es eine spezielle Annotation die angibt was die Funktion genau berechnet. Jede Funktion muss eine Return Annotation bekommen.

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

Die entsprechende Implementation:

```
function Inc(X: Integer) return Integer is
begin
return X + 1;
end Inc;
```

4.3.2 Package Annotations

Diese Annoations geben Auskunft über globale Variablen eines Pakets.

Own Annotation

Gibt die globalen Variablen bekannt die zu diesem Paket gehören.

```
Syntax:
--# own mode ownVariable , mode ownVariable;
Example:
--# own X, Y, State;
```

Beispiel Leergut 4.1.1

```
package Leergut
   --# own Psumme: Money;
```

Initializes Annotation

Gibt bekannt welche globalen Variablen vom Paket initialisiert werden.

```
Syntax:
--# initializes ownVariable , ownVariable;
Example:
package Random_Numbers
--# own Seed;
--# initializes Seed;
```

Beispiel Leergut (siehe 4.1.1)

```
package Leergut
   --# own Psumme: Money;
   --# initializes Psumme;
```

4.3.3 Inherit Annotation

Will man auf Inhalte eines Pakets zugreifen muss man dies mit einer Inherit Annotation angeben, diese folgt unmittelbar nach der with-clause. Desweiteren muss man beachten das indirekt verwendete Pakete, die zB. durch das gewollte Paket verwendet werden, mitangeben werden müssen.

```
Syntax:
--# inherit packageName , packageName;

Example:
with Leergut;
--# inherit Leergut, Spark_IO;
```

Das gewünschte Paket Leergut verwendet selbst das Paket Spark_IO, deshalb muss es auch mit aufgeführt werden.

4.3.4 main_program Annotation

Diese Annotation kennzeichnet das Hauptprogramm und wurde schon hier 4.2 erläutert.

4.3.5 Hide Annotation

Wie bereits erwähnt kommt man manchmal nicht um die Verwendung von "Full Ada" herum, zum Beispiel wenn man auf Hardware oder Betriebssystemfunktionen zugreifen muss. Die Ausgabe von Text wäre ebenfalls so ein Fall. Dabei kommt einen die Hide Annotation zu Hilfe, mit ihr kann man subprogram bodies, package bodies und den package specification private part vor dem Examiner verstecken. Bei der Verwendung von Hide muss man sich allerdings

äußerst sicher sein, dass entweder garkein Fehler auftreten kann oder sollte es doch zu einen kommen(zB. Lese-/Schreibfehler) entsprechend reagiert wird.

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

4.3.6 Verification Annotations

Um die Korrektheit eines Programms zu überprüfen(Verification) werden vom Examiner Hypothesen aufgestellt. Manchmal reichen die aufgestellten Hypothesen aber nicht zum endgültigen Beweis aus.

Pre- und Post- condition Annotations

Um zusätzliche Hypothesen zu erzeugen kann man einerseits seinen Code mit IF-Abfragen anpassen oder aber man stellt pre- und post- condition annotations. Mit ihnen trifft man Annahmen wie sich Variablen vor und nach dem Aufruf eines Unterprogramms verhalten/verändern. Dabei sollte man aber vorsichtig sein, falsche Annahmen über Vor- und Nachbedingungen können zu unbeweisbaren Hypothesen führen.

```
Syntax:
--# pre predicate;
--# post predicate;
Examples:
procedure Inc(X: in out T)
--# derives X from X;
--# pre X < T'Last;
```

```
...
procedure Exchange(X, Y: in out Float)
--# derives X from Y &
--# Y from X;
--# post X = Y~ and Y = X~;
```

Mit `Variablename~` gibt man an, dass der Wert der Variable vor dem Aufruf gemeint ist.

Assert Annotation

Assert Annotations dienen dazu Schleifen zu "cutten", da es sich bei Schleifen um zyklische Strukturen handelt, wir aber sicherstellen müssen, dass unser Programm nur 1 Eingang und eine endliche Zahl Ausgänge hat fügt der Examiner per Default vor jede Schleifeniteration eine Default Assertion die standardmäßig überprüft ob die verwendeten Variablen auch in ihren Grenzen liegen. Sollte diese Default Assertion aber nicht ausreichen kann man seine eigene definieren.

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

```
Example:
loop
   --# assert n in list_index;
    for i in Natural range 1 .. (n-1)
    loop
    --# assert n in list_index;
      if (lst(i) > lst(i+1) and i+1 <= n ) then
        tmp := lst(i);
        lst(i) := lst(i+1);
        lst(i+1) := tmp;
      end if;
    end loop;
    if (n > 1) then n := n - 1; end if;
   end if;
   exit when n = 1;
end loop;
```

5 Entwurf I: Modelierung einer Bank

Nun wollen wir ein SPARK konformes Programm schreiben. Als Beispiel nehmen wir uns hier eine Bank vor.

Folgende Anforderung sollte unsere Bank haben:

- es gibt Bankkonten(accounts)
- jedes Konto hat eine eindeutige ID
- jedes Konto besitzt einen Kontostand(balance), welcher nicht unter -30 Euro fallen darf(Dispo)
- zwischen den Konten kann Geld transferiert werden
- Informationen zu ID und Kontostand können abgefragt werden

5.1 Bank Framework

Musterlösung Bank in Ada

Annotations folgen

bank.ads

```
with Ada.Text_IO, Ada.Integer_Text_IO;

package bank is
   subtype ID is Natural;
   subtype money is Integer range -30..2147483647;

   type bank_account is record
      account_id : ID;
      balance : money;
   end record;

   procedure create_account(acc : out bank_account; bal : in money);

   procedure transfer(from_acc : in out bank_account;
                      to_acc   : in out bank_account;
                      amount: in Natural;
                      succ_trans : out boolean);

   procedure account_info(acc : in bank_account);

private
   id_counter : id := 0;

```

bank.adb

```
package body bank is

   procedure create_account(acc : out bank_account;
                            bal : in money)
   is
   begin
      acc := bank_account'(id_counter, bal);
      id_counter := id_counter + 1;
   end create_account;

   procedure transfer(from_acc : in out bank_account;
                      to_acc : in out bank_account;
                      amount: in Natural;
                      succ_trans : out boolean)
   is
   begin
      if (from_acc.balance >= money'First + amount
          and to_acc.balance <= money'Last - amount) then
         from_acc.balance := from_acc.balance - amount;
         to_acc.balance := to_acc.balance + amount;
         succ_trans := true;
       else
         succ_trans := false;
       end if;
   end transfer;

   procedure account_info(acc : in bank_account) is
   begin
     Ada.Text_IO.Put("ID: ");
     Ada.Integer_Text_IO.Put(acc.account_id, 10);
     Ada.Text_IO.Put_Line("");
     Ada.Text_IO.Put("Balance: ");
     Ada.Integer_Text_IO.Put(acc.balance, 10);
     Ada.Text_IO.Put_Line("");
   end account_info;

end bank;
```

Zur Überprüfung transferieren wir etwas Geld zwischen 2 Konten.

bank_main.adb

```
1  with bank;
2
3  procedure bank_main
4  is
5     acc1 : bank.bank_account;
6     acc2 : bank.bank_account;
7
8     transfer_ok : boolean;
9  begin
10    bank.create_account(acc1,2000);
11    bank.create_account(acc2,-15);
12
13    --Ada.Text_IO.Put_Line("From");
14    bank.account_info(acc1);
15    --Ada.Text_IO.Put_Line("To");
16    bank.account_info(acc2);
17    --Ada.Text_IO.Put("Amount: ");
18    --Ada.Integer_Text_IO.Put(30, 10);
19    --Ada.Text_IO.Put_Line("");
20
21    bank.transfer(acc1,acc2, 30, transfer_ok);
22
23    if(transfer_ok) then
24       --Ada.Text_IO.Put_Line("Money successfully transfered");
25       bank.account_info(acc1);
26       bank.account_info(acc2);
27    end if;
28
29 end bank_main;
```

5.2 Vorbereitung zur Untersuchung des Codes

Nun fügen wir Stück für Stück unsere Annotations hinzu, es wird aber auch durchaus empfohlen die Annotations gleich beim Entwerfen des Programms zu setzen. Um dann diese auf Konformität zu überprüfen nutzen wir den SPARK Examiner(siehe 6.1).

Diesen benutzen wir folgendermaßen **spark filename**

Um nun unsere 3 Dateien auf Konformität zu untersuchen, braucht der Examiner noch eine Datei in der unsere Packete mit Body- und Specification-Pfad gelistet sind. Würden wir beispielsweise bank.adb untersuchen, würde der Examiner einen Fehler melden:

`Semantic Error    : 11: There is no package declaration for bank.`

Deshalb müssen wir ihm mitteilen wo sich sämtliche Dateien von denen unsere zu untersuchende abhängt befinden. Dies tut man mit einer sog. Index-File, diese kann man manuell schreiben, wir können uns aber eines Tools welches zu SPARK gehört bedienen **sparkmake**. Bevor wir es benutzen müssen wir aber noch unser Hauptprogramm kennzeichnen, damit die Index-File korrekt erstellt werden kann. Wir fügen also −−# main_program; vor Zeile 3 in bank_main.adb. Dann rufen wir `sparkmake bank_main.adb` aus und erhalten eine bank_main.idx.

bank_main.idx

```
bank      specification is in *PFAD*\bank.ads

bank      body          is in *PFAD*\bank.adb

bank_main main_program  is in *PFAD*\bank_main.adb
```

Manchmal findet **sparkmake** nicht alle Pakete, dann muss man manuell nachbessern.

5.3 Untersuchung bank.ads

Die Index-File können wir nun dem Examiner übergeben und beginnen mit der Untersuchung von bank.ads.

```
spark -index_file=bank_main.idx bank.ads
```

Der Output, welcher ebenfalls in einer Reportfile(*.rep) gespeichert wird, sieht folgendermaßen aus:

spark.rep

```
Line
   1  with Ada.Text_IO, Ada.Integer_Text_IO;
           ^1           ^2
--- (  1)  Warning           :  1: The identifier Ada is either undeclared or not
           visible at this point.
--- (  2)  Warning           :  1: The identifier Ada is either undeclared or not
           visible at this point.

  15     procedure create_account(acc : out bank_account; bal : in money);
                    ^3
*** (  3)  Semantic Error    :154: The subprogram or task body create_account does
           not have an annotation.

  20     procedure transfer(from_acc : in out bank_account;
                            to_acc : in out bank_account;
                            amount : in Natural;
                            succ_trans : out boolean);
                    ^4
*** (  4)  Semantic Error    :154: The subprogram or task body transfer does not have
           an annotation.

  24     procedure account_info(acc : in bank_account);
                    ^5
*** (  5)  Semantic Error    :154: The subprogram or task body account_info does not
           have an annotation.
```

```
  28     id_counter : id := 0;
         ^6
*** (  6)  Semantic Error    :151: The variable id_counter does not occur either in
           the package own variable list or as a refinement constituent.
```

Wir erhalten zunächst 2 Warnungen, weil die 2 Ada Packete die wir zur Textausgabe benutzen, nicht in der Index-File stehen. Diese können wir aber bewusst ignorieren, da sie ohnehin programmkritischen Full Ada Code enthalten, d.h. wenn wir nicht umhinkommen Text ausgeben zu müssen, müssen wir den Code der die Inhalte dieses Packetes benutzt verstecken. Darauf kommen wir zurück wenn wir bank.adb untersuchen.

Die nächsten Warnungen

```
Semantic Error    :154: The subprogram or task body account_info does not have an annotation.
```

beziehen sich darauf dass keine unserer Unterprogramme mit mind. einer Derives Annotation ausgestattet sind.
Wir beginnen mit: `procedure create_account(acc : out bank_account; bal : in money);`
Die Derives Annotaion kennzeichnet den Informationsfluss, wie fließen nun die Informationen in create_account? Zum einen erstellen wir einen neuen Account acc aus einer ID(id_counter) und einem Kontostand(bal), zum anderen erhöhen wir den id_counter um eins so dass nachfolgende Paktete eine einzigartige ID erhalten. Die neue ID geht also aus sich selbst hervor. Nun können wir unsere Annotation folgendermaßen formulieren:

```
--# derives id_counter from * &
--#         acc from id_counter,bal;
```

Da id_counter eine globale Variable des Packetes ist und sie hier in unserer Derives Annotaion verwendet wird, wird eine Global Annotation fällig um id_counter sichtbar zu machen, ansonten

würde der Examiner meckern:

```
Semantic Error    :753: The identifier id_counter is either undeclared or
           not visible at this point. This identifier must appear in a
           preceding legal global annotation or formal parameter list.
```

Da die ID inkrementiert wird, muss sie sowohl lesend als auch schreibend verfügbar sein.

Das Endergebnis sollte folgendermaßen aussehen:

```
procedure create_account(acc : out bank_account; bal : in money);
--# global in out id_counter;
--# derives id_counter from * &
--#         acc from id_counter,bal;
```

Nun widmen wir uns:

```
procedure transfer(from_acc : in out bank_account;
                       to_acc   : in out bank_account;
                       amount: in Natural;
                       succ_trans : out boolean);
```

Zu allererst wird keine globale Variable verwendet, deshalb widmen wir uns der Derives Annotation. Wieder stellen wir uns die Frage wie die Informationen fließen. Wir können es eigentlich auch an den Parameter Modi schon erahnen, nämlich das alle OUT-Parameter von irgendwelchen IN-Parametern derived werden, d.h. from_acc, to_acc, succ_trans stehen auf jedenfall links von **from** und from_acc,to_acc,amount stehen irgendwo rechts davon. Betrachten wir nun unsere OUT-Kandidaten einzeln. Bei einer Überweisung wird from_acc Geld in Höhe von amount verlieren, dieses Geld wird aber nur übertragen falls das Konto von to_acc keinen Integeroverflow hat, deshalb hängt es auch von diesem ab. Und natürlich hängt die Überweisung auch von from_acc selbst ab, denn wenn nicht genug Geld auf dem Konto ist, findet keine Überweisung statt. Wir schreiben also:
`--# derives from_acc from from_acc, to_acc, amount;`
Im Umkehrschluss, hängt to_acc von den gleichen Faktoren ab:
`--# derives to_acc from from_acc, to_acc, amount;`
Wir können deshalb auch verkürzt für beide Zeilen schreiben:
`--# derives from_acc, to_acc from from_acc, to_acc, amount;`
Desweiteren haben wir noch eine Variable die kennzeichnet, ob die Übertragung aufgrund der oben genannten Faktoren erfolgreich war(succ_trans). Deshalb können wir diese ebenfalls von from_acc,to_acc,amount ableiten:

```
--# derives succ_trans from from_acc,to_acc, amount;
```

Alles zusammengefasst:

```
procedure transfer(from_acc : in out bank_account;
                   to_acc   : in out bank_account;
                   amount: in Natural;
                   succ_trans : out boolean);
--# derives from_acc, to_acc, succ_trans from from_acc, to_acc, amount;
succ_trans
```

Alternativ könnte man auch schreiben:

```
procedure transfer(from_acc : in out bank_account;
                   to_acc   : in out bank_account;
                   amount: in Natural;
                   succ_trans : out boolean);
--# derives from_acc   from from_acc, to_acc, amount &
--#         to_acc     from from_acc, to_acc, amount &
--#         succ_trans from from_acc, to_acc, amount;
```

Unser letztes Unterprogramm ist

```
procedure account_info(acc : in bank_account);
```

Es gibt mithilfe von Ada.Text_IO und Ada.Integer_Text_IO Informationen zum gewünschten Account aus. Da diese Pakete nicht SPARK-konform sind und es aufgrund ihrer kritischen Inhalte, nämlich der Textausgabe, es auch nie sein können, können wir auch nicht genau sagen wo unsere Kontoinformation hinfließt, wir wissen nur dass sie vom Parameter acc abhängt. Hier kommt uns der Null Derive zur Hilfe, mit ihm können wir angeben das wir die Information aus acc importieren, aber nicht sagen können wohin diese Exportiert wird:

```
procedure account_info(acc : in bank_account);
--# derives null from acc;
```

Nun feuern wir den Examiner nochmals ab und stellen fest, dass ihm id_counter unbekannt ist:

```
Semantic Error    :753: The identifier id_counter is either undeclared or
          not visible at this point. This identifier must appear in a
          preceding legal global annotation or formal parameter list.
```

Wir haben nämlich vergessen anzukündigen das er zum Paket gehört, dies tun wir mit der Own Annotaion:

```
package bank
--# own id_counter;
```

Da unser Paket den Counter mit 0 initialisiert müssen wir dies auch noch kennzeichnen:

```
package bank
--# own id_counter;
--# initializes id_counter;
```

Wir benutzen den Examiner nochmals und stellen fest, dass nur noch die Warnungen für die Ada-Pakete übrig sind, die wir ignorieren, weil wir Textausgaben nicht mit SPARK beweisen können, da sie auf Hardware zugreifen und so ein Fehler nicht auszuschließen ist, wir vertrauen darauf, dass im Ernstfall entsprechende Exceptions über das Problem geworfen werden. Unsere bank.ads sollte nun folgerndermaßen aussehen:

bank.ads

```
1  with Ada.Text_IO, Ada.Integer_Text_IO;
2
3  package bank
4  --# own id_counter;
5  --# initializes id_counter;
6  is
7     subtype ID is Natural;
8     subtype money is Integer range -30..2147483647;
9
10    type bank_account is record
11       account_id : ID;
12       balance : money;
13    end record;
14
```

```
15    procedure create_account(acc : out bank_account; bal : in money);
16    --# global in out id_counter;
17    --# derives id_counter from * &
18    --#         acc from id_counter,bal;
19
20    procedure transfer(from_acc : in out bank_account; to_acc : in out bank_account; amo
21    --# derives from_acc, to_acc, succ_trans from from_acc, to_acc, amount;
22
23    procedure account_info(acc : in bank_account);
24    --# derives null from acc;
25
26 private
27    id_counter : id := 0;
28
29 end bank;
```

5.4 Untersuchung bank.adb

Das Interface unserer Bank steht, nun müssen wir überprüfen ob der dazugehörige Code mit unseren Annotations konform ist. Desweiteren prüft der Examiner, ob unser Code den SPARK-Regeln entspricht, siehe 3.

```
spark -index_file=bank_main.idx bank.adb
```

Da die bereits erwähnten Ada-Pakete ignoriert werden können, setzen wir eine Hide Annotation ein um den Code von account_info zu verstecken.

```
   procedure account_info(acc : in bank_account) is
   --# hide account_info;
   begin
     Ada.Text_IO.Put("ID: ");
     Ada.Integer_Text_IO.Put(acc.account_id, 10);
     Ada.Text_IO.Put_Line("");
     Ada.Text_IO.Put("Balance: ");
     Ada.Integer_Text_IO.Put(acc.balance, 10);
     Ada.Text_IO.Put_Line("");
   end account_info;
```

Ein weiterer Aufruf des Examiners zeigt uns die bereits bekannten Warnungen über die Ada-Pakete und eine neue Warnung darüber, dass der Code von account_info versteckt wurde. Der Rest des Codes scheint SPARK konform zu sein.

5.5 Untersuchung bank_main.adb

Als letztes untersuchen wir das Hauptprogramm:

```
spark -index_file=bank_main.idx bank_main.adb
```

spark.rep

```
Line
   1  with bank;
             ^1
--- (  1)  Warning  :  1: The identifier bank is either undeclared or not
           visible at this point.
.
.
.
usw
```

Da wir das Paket Bank benutzen wollen, müssen wir dies dem Examiner mit der Inherit Annotation mitteilen:

```
with bank;
--# inherit bank;
```

spark.rep

```
5 error(s) or warning(s)

Line
   5  procedure bank_main
                ^1
*** (  1)  Semantic Error    :154: The subprogram or task body bank_main does not
           have an annotation.

  14     bank.create_account(acc1,2000);
         ^2,3
*** (  2)  Semantic Error    : 25: The identifier bank.id_counter (imported by called
           subprogram) is not visible at this point.
*** (  3)  Semantic Error    : 24: The identifier bank.id_counter (exported by called
           subprogram) is not visible at this point.
```

```
  15     bank.create_account(acc2,-15);
         ^4,5
*** (  4)  Semantic Error    : 25: The identifier bank.id_counter (imported by called
           subprogram) is not visible at this point.
*** (  5)  Semantic Error    : 24: The identifier bank.id_counter (exported by called
           subprogram) is not visible at this point.
```

Da unsere bank_main auch ein Unterprogramm ist benötigt sie auch die entsprechenden Annotations. Zum einen bekommen wir den Fehler das bank.id_counter nicht sichtbar ist. Da wir die ID indirekt benötigen muss diese auch angeben werden:

```
procedure bank_main
--# global in out bank.id_counter;
```

Nun zum Informationsfluss, unsere Main hat keine Parameter, d.h. Information kann nur von Außerhalb durch globale Variablen fließen. In der Tat verändern wir indirekt durch den Aufruf von create_account den Counter des Bank Paketes. Darum geben wir dies durch eine Derives Annotation an:

```
procedure bank_main
--# global in out bank.id_counter;
--# derives bank.id_counter from *;
```

Ein weiterer Aufruf vom Examiner zeigt uns, das wir vorerst fertig sind. Als Hinweis möchte ich noch hinzufügen, dass die auskommentierten Zeilen nur zur übersichtlicheren Überprüfung dienen. Wir können sie nicht "hiden", da man mit der Hide Annotation nur ganze Codeblöcke von Unterprogrammen verstecken kann. Wir könnten höchsten den brisanten Code in eine extra procedure packen und ihn dort verstecken.
Unsere konforme Main sollte nun so aussehen:

bank_main.adb

```
with bank;
--# inherit bank;

--# main_program
procedure bank_main
--# global in out bank.id_counter;
--# derives bank.id_counter from *;
is
   acc1 : bank.bank_account;
   acc2 : bank.bank_account;

   transfer_ok : boolean;
begin
   bank.create_account(acc1,2000);
   bank.create_account(acc2,-15);

   --Ada.Text_IO.Put_Line("From");
   bank.account_info(acc1);
   --Ada.Text_IO.Put_Line("To");
   bank.account_info(acc2);
   --Ada.Text_IO.Put("Amount: ");
   --Ada.Integer_Text_IO.Put(30, 10);
   --Ada.Text_IO.Put_Line("");

   bank.transfer(acc1,acc2, 30, transfer_ok);

   if(transfer_ok) then
      --Ada.Text_IO.Put_Line("Money successfully transfered");
      bank.account_info(acc1);
      bank.account_info(acc2);
   end if;

end bank_main;
```

Nun ist unser Gesamtes Programm SPARK konform und wir können uns an die Verifikation machen.

6 Analyse von SPARK-Programmen mit den SPARK-Tools

6.1 Examiner

Der SPARK Examiner hat folgender Aufgaben von denen einige kurz erläutert werden:

- Überprüft ob der Text(Code, Annotations) den SPARK Regeln entspricht
- Überprüft ob der auszuführende Code mit den Annotations übereinstimmt (zB. Zuweisungen $\hat{=}$ Derives)
- Erstellung von Verification Conditions VC (*.vcg) /vc
- Run Time Checks RTC, Overflow Checks /exp
- Path Functions /pfs

`Aufruf: spark [options]`

6.1.1 Verification Conditions

Um letztendlich unser Programmm beweisen zu können stellt der Examiner für jedes Unterprogramm und für jede Codezeile bzw. Statement in diesem Unterprogramm eine Verification Condition(VC) auf. Eine VC besteht aus Hypothesen und Konklusionen(Conditions) also Schlussfolgerungen. Mit Hilfe der Hypothesen sollen nun die Konklusionen bewiesen werden. Einige davon sind trivial zB. 5 ¡ 6 und können vom Examiner sofort bewiesen werden, andere muss man mit weiteren Sparktools bearbeiten um diese zu beweisen. VC werden mit dem Examiner flag /vc oder /exp erstellt, dabei erstellt /exp noch zusätzlich Run Time Checks. Generiert werden dabei *.vcg Dateien und zwar für jedes Unterprogramm eine. Desweiteren wird eine *.fdl Datei erstellt welche Deklarationen über Variablen(zB natural__last) und Prozeduren enthält. Und es gibt dann noch eine Proof Rules File *.rls welche Regeln enthält die auf die VCs angewendet werden dürfen(zB transfer_rules(7): natural__first may_be_replaced_by 0.) Die VC für die erste Zeile von transfer sieht so aus:

```
procedure_transfer_1.
H1:    true .
H2:    fld_balance(from_acc) >= money__first .
H3:    fld_balance(from_acc) <= money__last .
H4:    fld_account_id(from_acc) >= id__first .
H5:    fld_account_id(from_acc) <= id__last .
H6:    fld_balance(to_acc) >= money__first .
H7:    fld_balance(to_acc) <= money__last .
H8:    fld_account_id(to_acc) >= id__first .
H9:    fld_account_id(to_acc) <= id__last .
H10:   amount >= natural__first .
H11:   amount <= natural__last .
        ->
C1:    money__last - amount >= integer__base__first .
C2:    money__last - amount <= integer__base__last .
C3:    money__first + amount >= integer__base__first .
C4:    money__first + amount <= integer__base__last .
```

Generiert wurde sie mit: `spark /vc -index_file=bank_main.idx bank.adb`

6.1.2 Pfadgenerierung

Wenn man möchte kann man auch Paths, also Pfade die durch ein Unterprogramm genommen werden können generieren.

`spark /pfs -index_file=bank_main.idx bank.adb`

Diese befinden sich dann in *.pfs Dateien und zeigen wie durch ein Unterprogramm traversiert werden kann. Das ist vielleicht interessant wenn man viele verschachtelte Schleifen mit kompliziereten IF-ELSE-Verzweigungen hat und man seinen Algorithmus besser nachvollziehen will. Dennoch ist die Erstellung von Paths zum Beweis eine Programmes nicht nötig. sondern bietet einfach nur eine zusätzliche Untersuchungsmöglichkeit.

Für traverse sehen die Pfade vollgendermaßen aus:

```
Statement: start      1 successor(s)
  Successor statement:  finish.
    Path  1
      Traversal condition:
 1:    fld_balance(from_acc) >= money__first + amount .
 2:    fld_balance(to_acc) <= money__last - amount .
      Action:
        from_acc := upf_balance(from_acc, fld_balance(
           from_acc) - amount) &
        to_acc := upf_balance(to_acc, fld_balance(to_acc) +
           amount) &
        succ_trans := true .
    Path  2
      Traversal condition:
 1:    not ((fld_balance(from_acc) >= money__first + amount) and (
           fld_balance(to_acc) <= money__last - amount)) .
      Action:
        succ_trans := false .
```

6.1.3 Run Time und Overflow Checks

Zu empfehlen ist den Examiner zur VC-Genrierung mit dem /exp Flag zu benutzen. Dabei fügt der Examiner an kritischen stellen Assertions ein welche zur Laufzeit einen Fehler generieren wenn zB. Variablen überlaufen oder generell nicht in ihren Grenzen liegen. Das ermöglicht den Examiner zusätzliche VCs zu generieren die ,wenn sie bewiesen werden, sicherstellen dass das jeweilige Unterprogramm frei von Overflows ist. Dabei ist zu beachten das bei /exp nur Intergeroverflows betrachtet werden, will man auf Floats testen muss man das Flag /realrtcs in Verbindung mit /exp benutzen. Hierbei muss aber besondere Vorsicht gelten, da Real RTCs immer approximiert werden, der Examiner versucht dann die Floatwerte als Brüche in den VCs darzustellen. Will man Realtypen verwenden muss man diese entweder als Subtyp mit einer Range von Float ableiten oder als neuen Typ mit Digits und Range definieren.

6.2 Simplifier

Der Simplifier ist ein mächtiges Tool, welches die vom Examiner erstellen VCs vereinfacht. Dazu nutzt er ein umfangreiches mathematisches Regelwerk und die zusätzlichen Regeln der *.rls Dateien. Ersteres kann man auch in Checker_Rules.pdf nachlesen welches mit der SPARK-Distrubition kommt(siehe 4). Der Simplifier kann i.d.R. alle VCs lösen die man ihm vorsetzt, da er sie zu trivialen Aussagen reduziert.Sollte er mal etwas nicht lösen können, liegt es entweder an falschen Hypothesen die zu unlösbaren Schlussfolgerungen führen, wahrscheinlich entstanden durch falsche Pre- und Post-Conditions, oder an nicht vorhandenen Definitionen von Typgrenzen, wie zB. Integer. Da diese vom Compiler/Architektur abhängen müssen sie entweder als neue Regel in die *.rls Datei eingetragen werden oder als feste Typdefinition im Code deklariert werden.

Als Output erstellt der Simplifier Simplified Verification Conditions *.siv.

```
sparksimp
```

Seine Options sind nicht weiter spannend, man kann teilweise die Reihenfolge der Simplikation bestimmen und Logs generieren lassen. Er simplifiziert dabei immer alle *.vcg im aktuellem Verzeichnis und Unterverzeichnissen. Das /a Flag ist realativ nützlich, da es Timestamps nicht beachtet, d.h. bereits simplifizierte *.vcg werden nochmals betrachtet.

6.3 POGS Proof Obligation Summerizer

Dieses kleine Tool erstellt einen Bericht in Form einer *.sum Datei über bereits bewiesene Teile des Programms und welche es noch zu beweisen gilt. Einfach im gewünschten Verzeichnis ausführen:

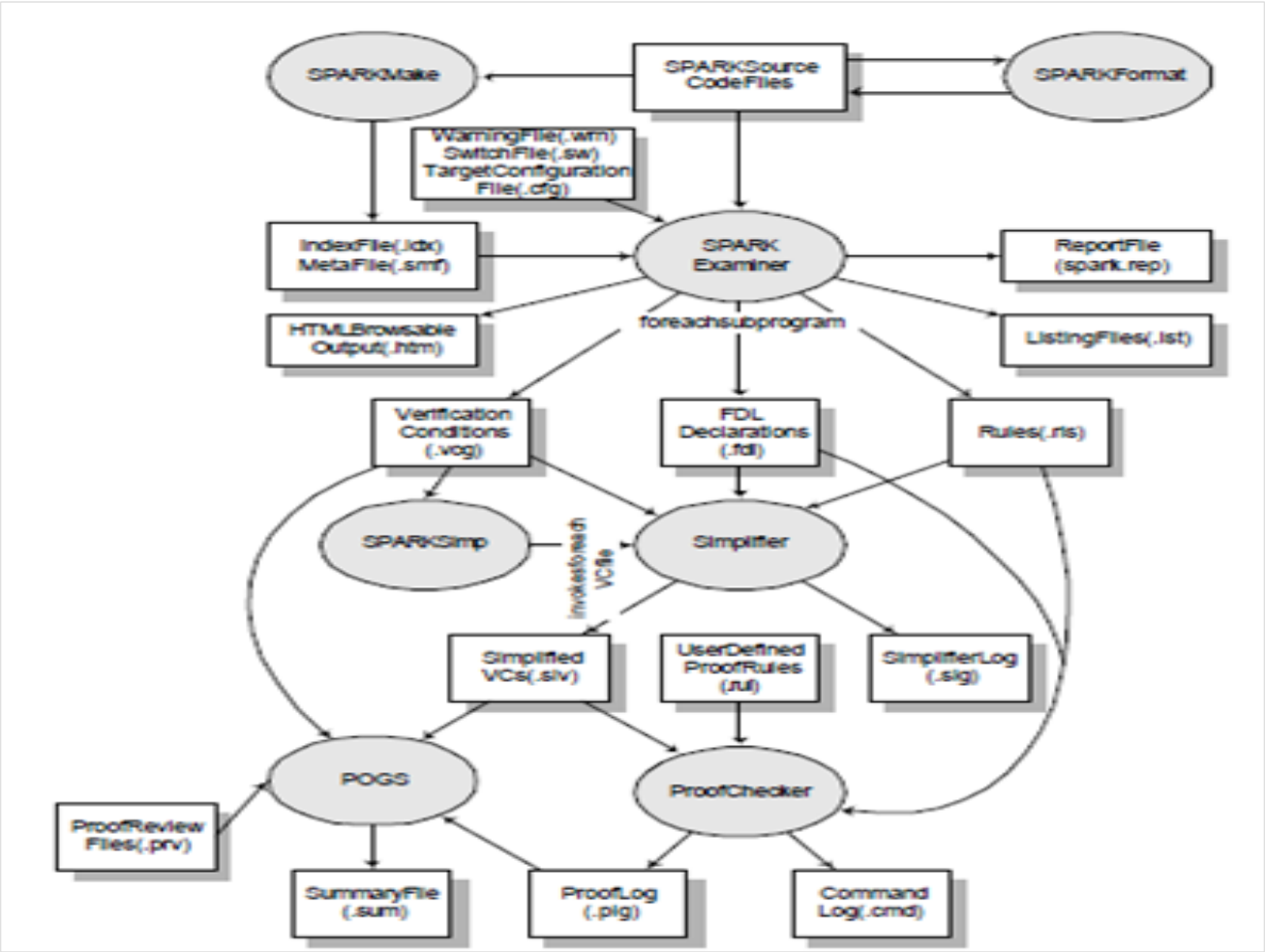
```
pogs
```

6.4 Proof Checker

Der Proof Checker ist ein interaktive Beweiskonsole, die zum Einsatz kommt, sollte der Simplifier doch einmal nicht ausreichen. Mit ihm kann man *.vcg und *.siv öffnen und sie mittels einer Konsole über eine Reihe von Befehlen Hypothesen/Konklusionen ableiten und umformen. Man kann ihn ebenfalls gesondert unter `http://libre.adacore.com/libre/download/` herunterladen. Ich werde ihn als folgenden Gründen hier aber vorerst nicht weiter behandeln: Erstens: Ich habe ihn noch nie zum Beweis eine Programmes verwendet, sondern lediglich ein paar Übungsaufgaben mit ihm gelöst. Zweitens: Er ist schon etwas älter(Ende der 90er) und ist sehr verbugt, er reagiert zB. manchmal nicht auf Commandos. Drittens: Wer sich näher mit ihm befassen möchte ihm liegt eine Umfangreiche Übungsserie bei, wenn man ihn runterlädt.

6.5 Tool Übersicht

Hier eine kleine Übersicht wie die Tools von SPARK zusammenhängen:



7 Entwurf II: Verifikation der Bank

Zur abschließenden Verifikation generieren uns zunächst unsere VCs mit Run Time Checks:

```
spark /exp -index_file=bank_main.idx bank_main.adb
```

```
spark /exp -index_file=bank_main.idx bank.adb
```

Nun lassen wir den Simplifier laufen und kucken was wir alles vereinfachen können:

`sparksimp`

Wir lassen uns alles mit `pogs`

zusammenfassen und schauen uns die **bank.sum** an:

gekürzte bank.sum

```
VCs for procedure_create_account :
------------------------------------------------------------------------------
     |       |                      |  -----Proved In-----  |       |       |
 #   | From  | To                   | vcg | siv | plg | prv | False | TO DO |
------------------------------------------------------------------------------
 1   | start | rtc check @ 5        |     | YES |     |     |       |       |
 2   | start | rtc check @ 6        |     |     |     |     |       |  YES  |
 3   | start |    assert @ finish   | YES |     |     |     |       |       |
------------------------------------------------------------------------------


------------------------------------------------------------------------------
     |       |                      |  -----Proved In-----  |       |       |
 #   | From  | To                   | vcg | siv | plg | prv | False | TO DO |
------------------------------------------------------------------------------
 1   | start | rtc check @ 11       |     |     |     |     |       |  YES  |
 2   | start | rtc check @ 12       |     | YES |     |     |       |       |
 3   | start | rtc check @ 13       |     | YES |     |     |       |       |
 4   | start |    assert @ finish   | YES |     |     |     |       |       |
 5   | start |    assert @ finish   | YES |     |     |     |       |       |
------------------------------------------------------------------------------


------------------------------------------------------------------------------
     |       |                      |  -----Proved In-----  |       |       |
 #   | From  | To                   | vcg | siv | plg | prv | False | TO DO |
------------------------------------------------------------------------------
 1   | start | rtc check @ 14       |     | YES |     |     |       |       |
 2   | start | rtc check @ 15       |     |     |     |     |       |  YES  |
 3   | start | rtc check @ 25       |     |     |     |     |       |  YES  |
 4   | start |    assert @ finish   | YES |     |     |     |       |       |
 5   | start |    assert @ finish   | YES |     |     |     |       |       |
------------------------------------------------------------------------------

                          -----------Proved By Or Using-----------
                    Total  Examiner Simp(U/R)  Checker Review False Undiscgd
Assert or Post:         5      5      0              0      0      0       0
```

```
Precondition check:      0      0      0              0      0      0       0
Check statement:         0      0      0              0      0      0       0
Runtime check:           8      0      4              0      0      0       4
Refinement VCs:          0      0      0              0      0      0       0
Inheritance VCs:         0      0      0              0      0      0       0
------------------------------------------------------------------------------
Totals:                 13      5      4              0      0      0       4  <<<
% Totals:                      38%    31%            0%     0%     0%     31% <<<
==================== End of Semantic Analysis Summary ========================
```

Beginnen wir mit `procedure_create_account`, wir sehen das in Zeile 6 Der RTC nicht aufgelöst werden konnte. Dazu schauen wir uns die `create_acount.siv` an:

gekürzte create_acount.siv

```
procedure_create_account_2.
H1:    bal >= - 30 .
H2:    bal <= 2147483647 .
H3:    id_counter >= 0 .
H4:    id_counter <= id__last .
H5:    integer__size >= 0 .
H6:    integer__first <= integer__last .
H7:    integer__base__first <= integer__base__last .
H8:    integer__base__first <= integer__first .
H9:    integer__base__last >= integer__last .
H10:   id__size >= 0 .
H11:   id__base__first <= id__base__last .
H12:   id__base__last >= id__last .
H13:   money__size >= 0 .
H14:   money__base__first <= money__base__last .
H15:   0 <= id__last .
H16:   id__base__first <= 0 .
H17:   money__base__first <= - 30 .
H18:   money__base__last >= 2147483647 .
       ->
C1:    id_counter + 1 <= id__last .
```

Wir bermerken dass nicht bewiesen werden kann unser Counter nie Größer als id__last wird, er kann also überlaufen. Wahrscheinlich haben wir vergessen diesen Fall in einer Abfrage abzufangen. Wir könnten jetzt einerseits eine Post-Kondition stellen und einfach annehmen das dieser Fall nie passieren wird:

```
procedure create_account(acc : out bank_account; bal : in money);
--# global in out id_counter;
--# derives id_counter from * &
--#         acc from id_counter,bal;
--# pre id_counter~ + 1 < id_counter;
```

Oder aber wir gehen auf Nummer sicher und ergänzen um eine IF-Abfrage:

```
procedure create_account(acc : out bank_account; bal : in money) is
begin
  if (id_counter <= Natural'Last -1) then
   acc := bank_account'(id_counter, bal);
   id_counter := id_counter + 1;
  else
   acc := bank_account'(0, bal);
  end if;
end create_account;
```

Wir verändern dabei den Initialwert von id_counter auf 1, und nehmen 0 quasi als Zustand für eine erschöpfte Anzahl an IDs und setzen ihn wenn ein Überlauf bevorsteht. Da wir den Code geändert haben, müssen wir unsere Tools nochmals durchlaufen lassen:

```
spark /exp -index_file=bank_main.idx bank.adb

sparksimp

pogs
```

gekürzte create_acount.siv

```
For path(s) from start to run-time check associated with statement of line 5:

procedure_create_account_1.
H1:    bal >= - 30 .
H2:    bal <= 2147483647 .
H3:    id_counter >= 0 .
H4:    id_counter <= id__last .
H5:    integer__size >= 0 .
H6:    integer__first <= integer__last .
```

```
H7:    integer__base__first <= integer__base__last .
H8:    integer__base__first <= integer__first .
H9:    integer__base__last >= integer__last .
H10:   natural__size >= 0 .
H11:   natural__base__first <= natural__base__last .
H12:   natural__base__last >= natural__last .
H13:   id__size >= 0 .
H14:   id__base__first <= id__base__last .
H15:   id__base__last >= id__last .
H16:   money__size >= 0 .
H17:   money__base__first <= money__base__last .
H18:   0 <= natural__last .
H19:   natural__base__first <= 0 .
H20:   0 <= id__last .
H21:   id__base__first <= 0 .
H22:   money__base__first <= - 30 .
H23:   money__base__last >= 2147483647 .
       ->
C1:    natural__last - 1 >= integer__base__first .
C2:    natural__last - 1 <= integer__base__last .

.
.
.

For path(s) from start to run-time check associated with statement of line 7:

procedure_create_account_3.
H1:    bal >= - 30 .
H2:    bal <= 2147483647 .
H3:    id_counter >= 0 .
H4:    id_counter <= id__last .
H5:    natural__last - 1 >= integer__base__first .
H6:    natural__last - 1 <= integer__base__last .
H7:    id_counter <= natural__last - 1 .
H8:    integer__size >= 0 .
H9:    integer__first <= integer__last .
H10:   integer__base__first <= integer__base__last .
H11:   integer__base__first <= integer__first .
H12:   integer__base__last >= integer__last .
H13:   natural__size >= 0 .
H14:   natural__base__first <= natural__base__last .
```

```
H15:   natural__base__last >= natural__last .
H16:   id__size >= 0 .
H17:   id__base__first <= id__base__last .
H18:   id__base__last >= id__last .
H19:   money__size >= 0 .
H20:   money__base__first <= money__base__last .
H21:   0 <= natural__last .
H22:   natural__base__first <= 0 .
H23:   0 <= id__last .
H24:   id__base__first <= 0 .
H25:   money__base__first <= - 30 .
H26:   money__base__last >= 2147483647 .
       ->
C1:    id_counter + 1 <= id__last .
```

Wie wir sehen hat sich unser Problem noch nicht aufgelöst, aber unsere IF-Abfrage ist trotzdem berechtigt, sie hat außerdem wie man sieht noch mehr Hypothesen erzeugt. Das Problem was wir hier haben liegt daran das die SPARK Tools unseren Compiler und unsere Architektur nicht kennen. Sie wissen nicht wie die Grenzen unserer Typen wie zB. Integer ausfallen. Man kann sich aber ganz einfach einmal `Integer'First` und `Integer'First` in einem Ada Programm auf seiner Maschine ausgeben lassen. Da ich ein 32bit System benutze sind liegt die Integerobergrenze(4Byte) bei 2147483647, die Untergrenze bei -2147483648. Wir können dem Simplifier diese Grenzen auf zweierlei Wege mitteilen entweder wir legen die Typdefinition schon mit dieser Range fest, dies funktioniert aber nicht bei Subytypen wie wir sie verwenden, weil diese arithmetisch immernoch in den übergeordneten Typen rechnen. Wir müssen also auf die zweite Methode zurückgreifen, nämlich die *.rls anzupassen und sie um die Typgrenzen zu erweitern. Ich musste hier 4 neue Regeln einführen, die eigentlich relativ selbserklärend sind: Zum einen lege ich mit 2 `may_be_replaced_by` Regeln meine Grenzen für Integer fest. Mit den letzten 2 `may_be_deduced` Regeln sage ich das Ende der Typen ID und Natural aus dem Ende von Integer geschlussfolgert werden dürfen, sie also das gleiche Ende besitzen.

create_acount.rls

```
create_accou_rules(1): integer__size >= 0 may_be_deduced.
create_accou_rules(2): integer__first <= integer__last may_be_deduced.
create_accou_rules(3): integer__base__first <= integer__base__last may_be_deduced.
create_accou_rules(4): integer__base__first <= integer__first may_be_deduced.
create_accou_rules(5): integer__base__last >= integer__last may_be_deduced.
create_accou_rules(6): natural__size >= 0 may_be_deduced.
create_accou_rules(7): natural__first may_be_replaced_by 0.
create_accou_rules(8): natural__first <= natural__last may_be_deduced.
create_accou_rules(9): natural__base__first <= natural__base__last may_be_deduced.
create_accou_rules(10): natural__base__first <= natural__first may_be_deduced.
create_accou_rules(11): natural__base__last >= natural__last may_be_deduced.
create_accou_rules(12): id__size >= 0 may_be_deduced.
create_accou_rules(13): id__first may_be_replaced_by 0.
create_accou_rules(14): id__first <= id__last may_be_deduced.
create_accou_rules(15): id__base__first <= id__base__last may_be_deduced.
create_accou_rules(16): id__base__first <= id__first may_be_deduced.
create_accou_rules(17): id__base__last >= id__last may_be_deduced.
create_accou_rules(18): money__size >= 0 may_be_deduced.
create_accou_rules(19): money__first may_be_replaced_by -30.
create_accou_rules(20): money__last may_be_replaced_by 2147483647.
create_accou_rules(21): money__base__first <= money__base__last may_be_deduced.
create_accou_rules(22): money__base__first <= money__first may_be_deduced.
create_accou_rules(23): money__base__last >= money__last may_be_deduced.
create_accou_rules(24): integer__last may_be_replaced_by 2147483647.
create_accou_rules(25): integer__first may_be_replaced_by -2147483648.
create_accou_rules(26): integer__last = natural__last may_be_deduced.
create_accou_rules(27): integer__last = id__last may_be_deduced.
```

Man kann auch User Defined Rules anlegen, dabei legt man eine **xxx.rlu** und/oder eine **yyy.rlu** an, wobei **xxx** das umschließende Verzeichnis ist, d.h. die definierten Regeln gelten für alle Unterprogramme in diesen Verzeichnis. **yyy** sei hingegen der Name eines Unterprogramms, diese Regeln gelten dann entsprechen für eine **yyy.vcg** Datei.
Aussehen könnte so etwas folgendermaßen:

```
my_rule(1): integer__last may_be_replaced_by 2147483647.
my_rule(2): integer__first may_be_replaced_by -2147483648.
my_rule(3): natural__last = integer__last may_be_deduced.
my_rule(4): natural__last = id__last may_be_deduced.
```

Alles sehr angenehm, man braucht nicht für jedes Unterprogramm jede dazugehörige *.rls ergänzen, und *.rlu werden auch nicht bei erneutem Simplifier-Aufruf überschrieben. Das Problem was ich aber mit User Defined Rules habe ist, dass sie einfach nicht entsprechend

funktionieren. Man kann zB. die selben Regeln in einer *.rlu anlegen, die man in einer *.rls ergänzn musste um die VCs vollständig zu beweisen, nur diesmal führen sie nicht zum Beweis der VCs. Ich denke das liegt daran das die User Defined Rules als **erstes** vom Simplifier angewendet werden, danach folgen erst die generierten *.rls. Es kommt also auf die Reihenfolge an, wir arbeiten also vorerst weiter mit der Methode die *.rls zu ergänzen.

Nun starten wir `sparksimp -a`, das /a Flag sorgt dafür das bereits simplifizierte VCs erneut bearbeitet werden.Tatsächlich konnten nun alle VCs in create_acount.siv bewiesen werden.

Nun wenden wir uns transfer.siv zu. Hier haben wir das selbe Problem, nämlich dass die Grenzen von Integer und Natural nicht gefunden wurden, wir ergänzen diese:

transfer.rls

```
transfer_rules(1): integer__size >= 0 may_be_deduced.
transfer_rules(2): integer__first <= integer__last may_be_deduced.
transfer_rules(3): integer__base__first <= integer__base__last may_be_deduced.
transfer_rules(4): integer__base__first <= integer__first may_be_deduced.
transfer_rules(5): integer__base__last >= integer__last may_be_deduced.
transfer_rules(6): natural__size >= 0 may_be_deduced.
transfer_rules(7): natural__first may_be_replaced_by 0.
transfer_rules(8): natural__first <= natural__last may_be_deduced.
transfer_rules(9): natural__base__first <= natural__base__last may_be_deduced.
transfer_rules(10): natural__base__first <= natural__first may_be_deduced.
transfer_rules(11): natural__base__last >= natural__last may_be_deduced.
transfer_rules(12): id__size >= 0 may_be_deduced.
transfer_rules(13): id__first may_be_replaced_by 0.
transfer_rules(14): id__first <= id__last may_be_deduced.
transfer_rules(15): id__base__first <= id__base__last may_be_deduced.
transfer_rules(16): id__base__first <= id__first may_be_deduced.
transfer_rules(17): id__base__last >= id__last may_be_deduced.
transfer_rules(18): money__size >= 0 may_be_deduced.
transfer_rules(19): money__first may_be_replaced_by -30.
transfer_rules(20): money__last may_be_replaced_by 2147483647.
```

```
transfer_rules(21): money__base__first <= money__base__last may_be_deduced.
transfer_rules(22): money__base__first <= money__first may_be_deduced.
transfer_rules(23): money__base__last >= money__last may_be_deduced.
transfer_rules(24): integer__last may_be_replaced_by 2147483647.
transfer_rules(25): integer__first may_be_replaced_by -2147483648.
transfer_rules(26): integer__last = natural__last may_be_deduced.
```

Ein weiteres `sparksimp -a` und auch dieses Unterprogramm wurde vollständig bewiesen.

Jetzt wenden wir uns noch bank_main.siv zu. Und wer hätte es gedacht: selbes Problem. Wir ergänzen bank_main.rls um:

```
bank_main_rules(24): integer__last may_be_replaced_by 2147483647.
bank_main_rules(25): integer__first may_be_replaced_by -2147483648.
bank_main_rules(26): natural__last = integer__last may_be_deduced.
```

Danach wider: `sparksimp -a` und `pogs`. Nun zeigt sich dass wir alle Teile unseres Bank Programmes beweisen konnten, damit ist es sogut wie Fehlerfrei von Bugs sofern wir die richtigen Typgrenzen eingesetzt haben. Dennoch besteht die Möglichkeit, dass bei unserer Textausgabe Fehler auftreten, dies nehmen wir aber in Kauf.

gekürzte bank.sum

```
Total VCs by type:
                          ----------Proved By Or Using-----------
                    Total  Examiner Simp(U/R)  Checker Review False Undiscgd
Assert or Post:         6       6       0          0      0      0        0
Precondition check:     0       0       0          0      0      0        0
Check statement:        0       0       0          0      0      0        0
Runtime check:         10       0      10          0      0      0        0
Refinement VCs:         0       0       0          0      0      0        0
Inheritance VCs:        0       0       0          0      0      0        0
================================================================================
Totals:                16       6      10          0      0      0        0
% Totals:                      38%     63%         0%     0%     0%       0%
===================== End of Semantic Analysis Summary ========================
```

7.1 Tipps zum auflösen von Verification Conditions

- Hypothesen durch pre- und post-conditions stärken
- RTC erkennen oft keine Typgrenzen, da diese compilerabhängig sind $->$
  - Eigene Typen declarieren
  - Proof Rules auf eigenen Compiler abstimmen
  - (natural__last may_be_replaced_by 2147483647.)
- Richtige Assertions treffen, Abfragen einbauen
- Proof Checker anwenden
- ggf. Absprache mit dem Entwicklerteam

7.2 Bank Framework mit Floats

Das Framework angepasst auf Floats:

bank.ads

```
1  with Ada.Text_IO, Ada.Float_Text_IO, Ada.Integer_Text_IO;
2
3  package bank
4  --# own id_counter;
5  --# initializes id_counter;
6  is
7     subtype ID is Natural;
8     subtype money is Float range -30.0..100000000.0;
9     subtype amounttype is Float range 0.0..100000000.0;
10
11    type bank_account is record
12       account_id : ID;
13       balance : money;
14    end record;
15
16    procedure create_account(acc : out bank_account; bal : in money);
```

```
17      --# global in out id_counter;
18      --# derives id_counter from * &
19      --#          acc from id_counter,bal;
20
21      procedure transfer(from_acc : in out bank_account;
22                         to_acc : in out bank_account;
23                         amount: in amounttype;
24                         succ_trans : out boolean);
25      --# derives from_acc, to_acc, succ_trans from from_acc, to_acc, amount;
26
27      procedure account_info(acc : in bank_account);
28      --# derives null from acc;
29
30  private
31      id_counter : id := 1;
32
33  end bank;
```

bank.adb

```
1   package body bank is
2
3      procedure create_account(acc : out bank_account;
4
5      begin
6        if (id_counter <= Natural'Last -1) then
7         acc := bank_account'(id_counter, bal);
8         id_counter := id_counter + 1;
9        else
10        acc := bank_account'(0, bal);
11       end if;
12     end create_account;
13
14     procedure transfer(from_acc   : in out bank_account;
15                        to_acc     : in out bank_account;
16                        amount     : in amounttype;
17                        succ_trans : out boolean) is
18     begin
19        if (from_acc.balance >= money'First + amount
20            and
21              to_acc.balance <= money'Last - amount) then
22           from_acc.balance := from_acc.balance - amount;
23           to_acc.balance := to_acc.balance + amount;
24           succ_trans := true;
25        else                                bal : in money) is
26           succ_trans := false;
27        end if;
28     end transfer;
29
30     procedure account_info(acc : in bank_account) is
31     --# hide account_info;
32     begin
33       Ada.Text_IO.Put("ID: ");
34       Ada.Integer_Text_IO.Put(acc.account_id, 10);
35       Ada.Text_IO.Put_Line("");
36       Ada.Text_IO.Put("Balance: ");
37       Ada.Float_Text_IO.Put(acc.balance,9,2,0);
38       Ada.Text_IO.Put_Line("");
39     end account_info;
40
41  end bank;
```

bank_main.adb

```
1   with bank;
2   --# inherit bank;
3
4   --# main_program
5   procedure bank_main
6   --# global in out bank.id_counter;
7   --# derives bank.id_counter from *;
8   is
9      acc1 : bank.bank_account;
10     acc2 : bank.bank_account;
11
12     transfer_ok : boolean;
13  begin
14     bank.create_account(acc1,2000.0);
15     bank.create_account(acc2,-15.0);
16
17     --Ada.Text_IO.Put_Line("From");
18     bank.account_info(acc1);
19     --Ada.Text_IO.Put_Line("To");
20     bank.account_info(acc2);
21     --Ada.Text_IO.Put("Amount: ");
22     --Ada.Integer_Text_IO.Put(30, 10);
23     --Ada.Text_IO.Put_Line("");
24
25     bank.transfer(acc1,acc2, 30.82, transfer_ok);
26
27     if(transfer_ok) then
28        --Ada.Text_IO.Put_Line("Money successfully transfered");
29        bank.account_info(acc1);
30        bank.account_info(acc2);
31     end if;
32
33  end bank_main;
```

Der Beweis geschieht auf gleiche Weise wie vorher, nur dass wir(neben Integer- und Naturalgrenzen) in transfer.rls und bank_main.rls noch die Grenzen für Floats hinzufügen, welche natürlich auch variieren können je nach Compiler/Architektur.

```
transfer_rules(29): float__first may_be_replaced_by -340282000000000000000000000000000000000.
transfer_rules(30): float__last may_be_replaced_by 340282000000000000000000000000000000000.

bank_main_rules(29): float__first may_be_replaced_by -340282000000000000000000000000000000000.
bank_main_rules(30): float__last may_be_replaced_by 340282000000000000000000000000000000000.
```

Leider ist mir keine verkürzte Schreibweise bekannt, folgendes geht jedenfalls **nicht**:

```
transfer_rules(29): float__first may_be_replaced_by 3.40282E+38.
transfer_rules(30): float__last may_be_replaced_by 3.40282E+38.

oder

transfer_rules(29): float__first may_be_replaced_by 3.40282**38.
transfer_rules(30): float__last may_be_replaced_by 3.40282**38.

funktionieren nicht!
```