Bernburg
Dessau
Köthen

Hochschule Anhalt
Anhalt University of Applied Sciences

emw
Fachbereich
Elektrotechnik, Maschinenbau
und Wirtschaftsingenieurwesen

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Engineering (B. Eng.)

Xufeng Li

Vorname Nachname

Elektro- und Informationstechnik, 2011, 4055536

Studiengang, Matrikel, Matrikelnummer

Thema: Robuste Implementierung von Vektoroperationen

Prof. Dr. Marc Enzmann

1. Prüfer(in)

Prof. Dr. Andrea Jurisch

2. Prüfer(in)

13. 02. 2015

Abgabe am

Selbstständigkeitserklärung

Hiermit erkläre ich, dass die Arbeit selbstständig verfasst, in gleicher oder ähnlicher Fassung noch nicht in einem anderen Studiengang als Prüfungsleistung vorgelegt wurde und keine anderen als die angegebenen Hilfsmittel in Quellen, einschließlich der angegebenen oder beschriebenen Software, verwendet wurden.

Köthen, 10.02.2015

Ort, Datum Unterschrift des Studierenden

Sperrvermerk

Sperrvermerk: ja [ ] nein [X]

Wenn ja: Der Inhalt der Arbeit darf Dritten ohne Genehmigung der/des (Bezeichnung des Unternehmens) nicht zugänglich gemacht werden. Dieser Sperrvermerk gilt für die Dauer von X Jahren.

Köthen, 10.02.2015

Ort, Datum Unterschrift des Studierenden

Inhaltsverzeichnis

1 Motivation und Zielsetzung

1.1 Einleitung in die Thematik

Wenn man numerische Berechnungen durchführt, entstehen gelegentlich Fehler. Normalerweise läuft ein Code in den problemlosen Fällen. Aber er verursacht sporadische Softwarefehler, die zu fehlerhaften Berechnungsergebnissen, schlimmer noch, Programmabstürzen führen können. Dabei liegt die Fehlerursache häufig nicht im Programm selber, sondern daran, dass numerische Zahlendarstellungen einen falschen Zustand aufweisen. Zum Beispiel keine Initialisierung, ein überschrittener Wertebereich oder eine überschrittene Auflösung.
Die Vektoroperation spielt eine wichtige Rolle in unserem Leben. Sie wird in vielen Bereichen häufig genutzt. Zum Beispiel für die Bearbeitung von Signalen und die Berechnung der Faltung. Daraus ergibt sich die Notwendigkeit, eine sichere und hohe Robustheit der Bibliothek von Vektoroperationen zu entwickeln, die Fehler minimieren und vermeiden kann.
Zuerst musste über eine fehlerfreie Software nachgedacht werden. Warum ist das für den Softwareentwurf sehr wichtig? Das Buch „Software Engineering“ (Ian Sommerville, 2001) bildete eine wichtige Ausgangsposition für das Verständnis und den Kenntniserwerb.

Ein guter Softwareprozess sollte die Entwicklung fehlerfreier Software zum Ziel haben. Fehlerfreie Software ist Software, die genau ihrer Spezifikation entspricht. Das bedeutet jedoch nicht unbedingt, dass sich die Software immer so verhalten wird, wie die Benutzer das erwarten. Die Spezifikation kann Fehler enthalten, die in der Software widergespiegelt werden, und die Benutzt könnten das System missverstehen oder falsch verwenden. Fehlerfreie Software ist daher für die Anzahl der Systemausfälle bedeutend und sollte in der Entwicklung kritischer Systems stets angestrebt werden. [2]

Um ein robustes und wenig fehlerhaftes Programm entwerfen zu können, muss zuerst das Programm in einer robusten Umgebung entworfen werden. Das heißt, man sollte eine stark typisierte Programmiersprache auswählen, die eine hohe Sicherheit gewährleistet.
Wenn man über die Sicherheitsstufe von Programmiersprachen diskutiert, kann man drei Sicherheitsstufen zusammenfassen. Die erste Stufe ist die Programmiersprache Spark, die eine ‚Correctness by Construction‘ Philosophie verfolgt. Sie kann durch die Definition von Vor- und Nachbedingung realisiert werden, die statisch und dynamisch Fehler während der Compilezeit und Laufzeit vermeidet. Dann gibt es die Sprache Ada. Sie kann die Fehler während der Compilezeit prüfen und statische Fehler vermeiden. Zum Schluss ist die Sprache,

es gibt keine Prüfung, bevor das Programm läuft. Das bedeutet, während des Programmlaufs können statische und dynamische Fehler passieren und dann wird das ganze Programm zerstört. Mit einer sicheren Sprache kann man ein robustes Programm aufbauen. Also sollte man das Prinzip der Fehleranalyse auch beachten.
Über dieses Prinzip konnte man auch in dem Buch „Software Engineering“ ((Ian Sommerville, 2001) nachlesen:

1. Bestimmung und Bewertung der Vermögenswerte: In diesem Schritt werden die Vermögenswerte (Daten und Programm) und die erforderlichen Mechanismen zu ihrem Schutz bestimmt. Beachten Sie, dass der Grad des erforderlichen Schutzes vom Vermögenswert abhängt.
2. Bedrohungsanalyse und Risikobewertung: In dieser Phase werden mögliche Bedrohungen der Systemsicherheit bestimmt und diese mit jedem von ihnen verbundenen Risiko eingeschätzt.
3. Bedrohungszuordnung: Erkannte Bedrohungen werden dem Vermögenswert zugeordnet, so dass jeder Vermögenswert eine Liste mit zugeordneten Bedrohungen erhält.
4. Technologieanalyse: Dieser Schritt umfasst die Bewertung der zur Verfügung stehenden Sicherheitstechnologien und ihrer Anwendbarkeit auf die ausgemachten Bedrohungen.
5. Spezifikation der Anforderungen an die Systemsicherheit: In dieser Phase werden die Anforderungen an die Systemsicherheit festgelegt. Die Spezifikation enthält, sofern nötig, eine explizite Aufzählung aller Sicherheitstechnologien, die verwendet werden können, um das System gegen die verschiedenen Bedrohungen zu schützen. [2]

Dann kann eine robuste Bibliothek der Vektoroperation und das Testprogramm entworfen werden. Es kann durch die Begrenzung der Vor- und Nachbedingung die Fehler exakt kontrollieren und die Prüfleistung sparen. Hier wurde die Programmiersprache Spark verwendet. Spark besitzt mehrere Vorteile gegenüber anderen Programmiersprachen. Wenn das Programm in Spark entworfen wird, bevor es läuft, wird die eingegebene Variable automatisch kontrolliert, ob der Wert in Ordnung ist. Deshalb kann Spark die statischen und dynamischen Fehler effektiv vermeiden. Spark ist sicher als die Programmiersprache, die nur das Defensiv Programm Prinzip verwendet. Durch die robuste Bibliothek kann man auch den Fehlerzustand bei der Vektoroperation vermeiden, um eine hohe Robustheit und Sicherheit der Vektoroperation und eine exakte Kontrolle der eingegebenen und ausgegebenen Werte zu realisieren.

1.2 Zielsetzung der Arbeit

Das Thema „Robuste Implementierung numerischer Vektoroperationen“ ist Gegenstand dieser Bachelorarbeit.
Um numerische Vektoroperationen robust implementieren zu können, muss man einen Code schreiben. Darin sollten nicht nur die Fehler der Vektoroperationen getestet, sondern auch eine Begrenzung der eingegebenen und ausgegebenen Werte entworfen werden. Die erste Aufgabe ist der Entwurf der Vor- und Nachbedingung in der Programmierumgebung Spark.

SPARK ist eine formal definierte Computerprogrammiersprache basierend auf der Ada Programmiersprache. Sie ist eine Softwareentwicklungstechnologie, die mit hoher Zuverlässigkeit konzipiert wurde. Die Vor- und Nachbedingung sind sehr wichtige Teile in Spark. Sie können durch die Angabe des beabsichtigten Verhaltens das Programm stärken. Deshalb ist die erste Aufgabe, dass einige Vor- und Nachbedingungen in SPARK programmiert werden, um die Erzeugung von Fehlern bei den Vektoroperationen zu vermeiden. Das heißt, es muss den Fehlern in der Quelle vorgebeugt werden.
Aber das Testprogramm kann mit Spark die Vor- und Nachbedingung, die schon programmiert wurde, nicht schreiben. Mit Hilfe von Ada kann das Testprogramm die Vor- und Nachbedingung, die in Spark entworfen wurde, schreiben.

Allgemein gesagt, die Aufgaben in dieser Arbeit sind folgendermaßen:

- Entwurf und Programmierung der Vor und Nachbedingung im Spark
- Entwurf und Programmierung der Testprogramm für alle Vor und Nachbedingung

Als Lösungskonzept soll in der Aufgabe eine Bibliothek über den Test der numerischen Vektoroperationen entwickelt werden. Zuerst muss man alle Vektoroperationen kennen. Der nächste Schritt ist die Analysierung der potenziellen Fehler bei den Vektoroperationen. Hier ist eine mathematische Analyse sehr wichtig. Mit Hilfe der mathematischen Grundlagen kann der Plan für den Entwurf der Vor- und Nachbedingung umgesetzt werden. Danach erfolgt der Entwurf der Hilfsfunktionen für die Vor- und Nachbedingung. Mit Hilfe der Verwendung der Hilfsfunktion in der Spezifikationsdatei können Fehler vermieden werden, und die Codes bleiben sauber. Am Ende erfolgt die Programmierung aller Vor- und Nachbedingungen für alle Vektoroperationen.

2 Theoretische Grundlagen

In diesem Kapital werden die Programmiersprache Spark, die Bedeutung der Vor- und Nachbedingung und die Wichtigkeit erklärt.

In diesem Programm bestehen die Vektoroperationen aus vier mathematischen Funktionen. Das sind Addition, Subtraktion, Multiplikation und Skalarprodukt zwischen zwei Vektoren. Die Aufgabe für jede Vektoroperation ist es, eine gültige Vor- und Nachbedingung zu entwerfen. Die entworfene Vor- und Nachbedingung wird zuerst durch den mathematischen Beweis bestimmt. Für manche Operationen, z. B. Scale, ist es eine Multiplikation zwischen einem Vektor und einem Faktor. Es ist mathematisch einfach zu analysieren, deshalb muss man nur wenige Vorbedingungen für diese Operation definieren. Aber für die Operation Skalarprodukt, sie besteht aus dem Additions- und Multiplikationsvorgang, ist sie im Mathematischen schwer zu analysieren. Um die eingegebenen Werte dieser Operation exakter kontrollieren zu können, müssen für diese Operation nicht nur eine, sondern auch zwei oder drei Vorbedingungen entworfen werden.

Zu Beginn wird die Programmiersprache Spark erklärt.

2.1 Vorstellung von Spark

In diesem Programmentwurf wurde die Programmiersprache Spark verwendet, weil Spark eine hoch typisierte Sprache ist und auch die höchste Sicherheitsstufe hat. In Spark können auch die Vor- und Nachbedingungen für jede Prozedur definiert werden. Es ist besser für dieses Thema, robuste Vektoroperationen zu entwerfen.

2.1.1 Besonderheit

Stark typisiert

Spark ist eine stark typisierte Sprache. Eine Programmiersprache, die stark typisiert ist, kann die Richtigkeit der Daten besser erkennen. Spark und Ada sind stärker und haben eine höhere Sicherheit, als andere Programmiersprachen wie C, C++, Java. Durch die starke Typisierung können einige Rechenfehler durch Compilierzeit und Laufzeit geprüft werden.

Modular

Spark bietet einen so genannten „Pakete“ - Mechanismus. Pakete können als separate Übersetzungseinheiten definiert werden, so dass eine sehr effiziente Zusammenstellung separater Modelle für den Entwickler zur Verfügung steht. Pakete werden in zwei Teile aufgeteilt. Das sind die Paketspezifikation und die Implementierung. Die Spezifikation von Paketen ist im Paket Daten und Unterprogramme definiert. Die Implementierung, so genannte

Bodys, stellt den Inhalt der Implementierung aller Unterprogramme dar. Diese Kombination zwingt die Programmierer dazu, einen Vertrag zu definieren, um die Schnittstelle in allen Modulen und in einem Paket verwenden zu können.

Objektorientierung

Ada unterstützt die erweiterbare Programmierung. Es unterstützt auch aktiv verschiedene Programmaufrufe.

Flexibilität

Die neueste Version von Spark, Spark 2014, bietet auch die Flexibilität der Konfiguration der Beschränkung der Programmiersprache auf jedes Projekt. Und sie erlaubt die gültige Veränderung der Run-time-Umgebung.

Der Code in Spark 2014 kann einfach mit Ada oder C kombiniert werden. Das bedeutet, ein neues System kann entwickelt und die alte Datenbank wieder verwendet werden.

Ausdrückbar

In Spark kann man die eigenen numerischen Typen definieren. Der Compiler kann dann bestimmen, wie diese Typen angemessen eingesetzt werden.

Z. B. type Spannungen is digit 12 range -1.0... 1.0;

Die obige Definition bezieht sich auf einen Typ mit Spannungen mit 12 Dezimalstellen Genauigkeit und einen gültigen Wertebereich von -1,0 bis 1,0. Wenn ein Fehler auftritt, gibt es zwei Möglichkeiten: statisch und dynamisch. Ein statischer Fehler passiert in der Compilezeit und ein dynamischer Fehler passiert in der Laufzeit. Im Folgenden sind Beispiele von Fehler aufgelistet.

- **Statischer Fehler**

454:46 value not in range of type "Real_Type" defined at vectors.ads:14

Weil der Wert von dem Vektor größer als die Grenze von "Real_Type" ist, entsteht dieser Fehler. Diese Fehler passieren in der Compilezeit, wenn man einen zu großen Wert, der schon übergelaufen ist dem Vektor zuweist. Wenn ein statischer Fehler auftaucht, stoppt der Compiler und das Programm läuft nicht.

- **Dynamischer Fehler**

Er passiert in der Laufzeit. Der dynamische Fehler ist wie Constraint_Error, Tasking_Error, Programm_Error. Hier soll ein Constraint_Error erzeugt werden. Dieser Fehler passiert, wenn man den Wert von LeftV und RightV nicht richtig zuweist, sodass das Ergebnis der Addition keinen Constraint_Error erzeugt. Z. B. wenn man 0.4*Real_Type'Last beiden Vektoren zuweist, bekommt man das Ergebnis, einen Wert von 0.8*Real_Type'Last. Er kann keinen Overflow_Error(Constraint_Error) erzeugen, deshalb tritt dieser Fehler auf. Wenn ein dynamischer Fehler auftaucht, führt dies nicht nur zum Stopp des Programms, sondern auch zur

Zerstörung des Programms. Deshalb ist ein dynamischer Fehler schlimmer als ein statischer Fehler.

2.1.2 Modulare Programmierung

Ein Modul ist eine Sammlung von Funktionen oder Klassen, die zur Realisierung einer bestimmten Funktionalität nötig sind. [3]

Je größer ein bestimmtes Projekt ist, desto mehr Programmierer sind in der Regel daran beteiligt. Eine Aufteilung eines großen Projekts auf verschiedene Programmierer ist eine gute Methode. Jeder Programmierer muss seinen eigenen Code schreiben, damit viele Module entstehen.

Die Modulare Programmierung ist ein Programmierparadigma. Es hat das Ziel, Programme in logische Module zu gliedern. Jede Aufgabe wird in kleinere, handlichere Teilaufgaben zerlegt und implementiert. Ein Modul packt seine Daten in einer internen Datenstruktur ein. Jedes Modul kann einzeln geplant, programmiert und getestet werden. Universelle Module können immer wieder verwendet werden. Alle Module können logisch miteinander verknüpft und zusammengesetzt werden.

Modula-2, Ada, Oberon und Komponente Pascal sind typische modulare Programmierungssprachen. In anderen Programmierungssprachen bieten sich Bibliotheken an, die die Modularisierung der Programmierungssprache imitieren können, wenn diese von der Sprache nicht unterstützt wird.

In einer objektorientierten Sprache, z. B. in Java, entspricht die Klasse, die nach außen sichtbar ist, einem Modul. Der gesamte Quelltext von einer Klasse ist eine zusammenhängende Einheit. Und eine Klasse in Java darf nicht über mehrere Bereiche des Quelltextes verteilt werden. In einer anderen Sprache wie C++ kann die Implementierung der Methode der Klasse auf mehrere Quelltestmodule verteilt werden, weil im C++ die Deklaration einer Klasse von ihrer Implementierung getrennt ist.

2.1.2.1 Paket

Ein Paket hat die Fähigkeit, die verschiedenen Objekte in einer gleichen Datei zusammenzustellen. Dieses Objekt kann ein Dateityp, Dateiobjekt, Unterprogramm und sogar ein anderes Paket sein. Ein Paket kann auch versteckt dargestellt werden. Die ausführlichen Inhalte von privaten Anteilen und Bodys sind nicht sichtbar für die externen Benutzer. Ein Paket kann auch einen initialisierten Anteil haben.

Z. B:

```
Package Stacks.Monitoring is
        Type Monitored_Stack is new Stacks.Stack with private;
         Procedure Clear(S: out Monitored_Stack);
        Function Top_Identity(S: in Monitored_Stack) return Integer;
end Stacks.Monitoring;
```

Anhand des obigen Beispiels wird deutlich, dass das Paket verschiedene Objekte in einer Datei zusammenfassen kann.

Ads und Adb-Datei sind sehr wichtig in Ada und Spark, weil alle Prozeduren, Funktionen und Pakete diese zwei Anteile besitzen müssen. Ads-Datei (Spezifikation) ist ähnlich wie eine Header-Datei von C(.h), sie besitzt die Deklaration. Eine Spezifikation ist übersichtlicher und meist leichter zu verstehen als eine komplexe Implementierung in Adb-Datei. Die Adb-Datei (Body) ist ähnlich wie eine C-Datei von C(.c). Sie besitzt den Inhalt einer ganzen Implementierung, zum Beispiel B Prozedur, Funktion und Unterprogramm. Die Namen der Ads-Datei und ihre entsprechende Adb-Datei müssen gleich sein. Beim Verbinden der Datei wurde 'With' und 'Use' verwendet, z. B. beim Beginn des Programms:

```
With Vectors.Filter_Parameters, Vectors.Standard_Vectors;
use Ada;
```

Mit 'With' und 'Use' wurden alle Typen, Funktionen und Prozeduren der Ads-Datei sichtbar gemacht. Wenn ein PaketA ein 'With' verwendet und mit PaketB verbunden wird, entsteht eine Abhängigkeit zwischen diesen beiden Paketen. Wenn PaketA abhängig von PaketB ist, darf PaketB nicht PaketA verwenden, sonst taucht eine Fehler 'Circular Abhängigkeit' auf. Es gibt noch eine Methode, ein Paket mit einem anderen Paket zu verbinden. Dies kann durch die Erzeugung des Kindpakets realisiert werden. Das Kindpaket kann alle Typ von seinen Eltern erben und verwenden. Z. B. in diesem Programm heißt das Hauptpaket „Vector". Wenn man ein Kindpaket definieren möchte, kann man das Paket 'Vector.ops' erzeugen. Dann verbindet sich das Paket 'Vector.ops' automatisch mit 'Vector' und kann alle Typen, die im Hauptpaket definiert wurden, verwenden.

2.1.2.2 Unterprogramm

Ein Unterprogramm bedeutet eine Folge von Arbeiten und es funktioniert, wenn das Unterprogramm aufgerufen wird. Prozedur und Funktion sind zwei wichtige Anteile des Unterprogramms. Diese zwei Typen von Unterprogrammen können in einem Paket zusammenge-

schrieben werden und besitzen auch beide Ads- und Adb-Dateien. Nachfolgend sollen ausführlich diese Erkenntnisse erläutert werden.

2.1.2.2.1 Prozedur

Eine Prozedur beginnt mit seiner Spezifikation, die eine Liste von Parametern besitzt. Eine lokale Variable kann zwischen begin und end deklariert werden und die Operation passiert auch zwischen begin und end.
Hier ist ein Beispiel für die Struktur einer Prozedur aufgeführt.

```
Procedure Add (Result: out Basic_Vector;
                    Left, Right: in Basic_Vector);
      is
            LeftV, RightV: Real_Type;
            begin
                  Inhalt von Implementierung
end Add ;
```

Eine Prozedur kann Parameter in drei verschiedenen Modi enthalten. Sie sind in, out und in-out. Sie unterschieden sich durch die Richtung des Transports der Parameter. Beim in, dieser Parameter ist eine Konstante und diese Konstante wird durch den Wert des aktuell verbundenen Parameters initialisiert. Beim in – out, dieser Parameter ist eine Variable und wird durch den aktuellen Parameter initialisiert. Es erlaubt das Auslesen und Einschreiben beider von dem Wert des aktuell verbundenen Parameters. Beim out, dieser Parameter ist eine nicht initialisierte Variable, sie erlaubt die Aktualisierung des verbundenen aktuellen Parameters.

```
Z. B: Procedure Add (Result: out Basic_Vector;
                          Left, Right: in Basic_Vector);
```

Left und Right wird ein Wert eingegeben und dann wird Result zurückgegeben. Man kann dies auch folgendermaßen zusammenfassen: Wenn man einen Parameter importieren möchte, muss man die Modi in oder in-out verwenden. Wenn man einen Parameter exportieren möchte, muss man die Modi in- out oder out verwenden.

2.1.2.2.2 Funktion

Eine Prozedur beginnt mit seiner Spezifikation, die eine Liste von Parametern besitzt. Lokale Variable kann zwischen begin und end deklariert werden und die Operation passiert auch zwischen begin und end. Diese Funktion ist ähnlich wie die Form der Prozedur, aber sie startet mit der Wort Funktion und das Ergebnis wird nach der Liste der Parameter zurückgegeben.

Z. B: funktion Max(I,J :Interger) return Integer is
    Result: Interger;
     begin
          Inhalt von Implementierung
     return Result;
    end Max;

Das obige Beispiel ist die Struktur der Funktion in einer Adb-Datei. Eine Funktion kann nur In-Parameter bekommen und muss mindestens ein Ergebnis zurückgeben. Dabei wird auch der Rückgabewert angegeben, der von der Funktion an die Aufrufstelle zurückgegeben wird: return betrag.
Aber es ist auch wichtig zu wissen, bei der Funktion kann nur ein singuläres Ergebnis zurückgegeben werden.

2.1.3 Typen

Mit Typen hat der Programmierer die Möglichkeit, sich eigene Wertebehälter zu erstellen. Die Typen sind in Abbildung 1 dargestellt.

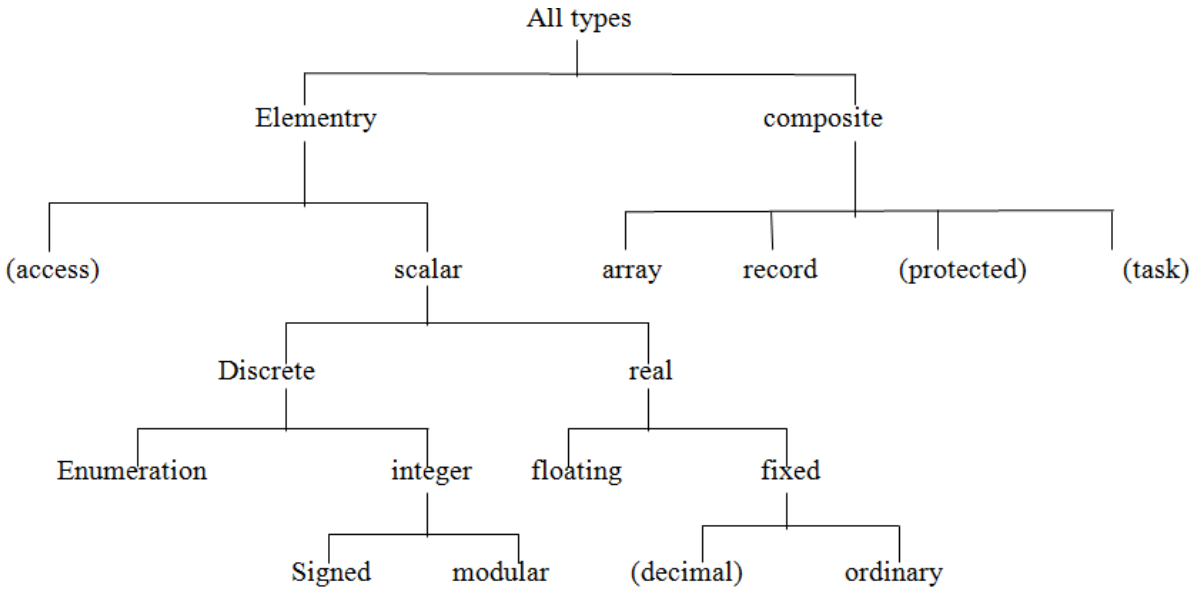

Abbildung 1: 3.1 SPARK type Quantisierung [1]

2.1.3.1 Arrays

Bei der Deklaration von Arrays gibt es zwei Typen. Sie sind constrained array und unconstrained array. Wenn man ein constrained array definieren möchte, müssen die obere und untere Grenze des Index Typ von Array festgestellt werden.

Z. B: subtype Index is Integer (range (0..10));
Type Row is array(Index) of Integer;

Der Vektor wird als Array Typen definiert, und der Index des Vektors muss zwischen 0 und 10 liegen.
Ein anderer Typ von Arrays ist unconstrained Array. Manche Programmiersprache bietet auch eine Unterstützung des unconstrained Array. Wenn man ein unconstrained Array defi-

niert, nutzt man ein Box Symbol (<>). Der Index eines unconstrained Array hat keine bestimmte Grenze, deshalb verwendet man das Box Symbol (<>), um die Grenze zu ersetzen.

Z. B type Tuple is array (Integer range<>) of Real
    Subtype Triple_Index is Integer range(1…3) of Real_Type;
    Subtype Triple is Tuple(Triple_Index) of Real_Type;

Wenn man auf ein bestimmtes Element von Array zugreifen möchte, kann man eine runde Klammer nutzen, z. B:Vector(2). Man kann nicht nur ein Element, sondern auch einen Bereich von Array wählen und darauf zugreifen.
Z. B Vector(2…5).

2.1.3.2 Private Typen

Eine Paket versorgt einen abstrakten Datentyp. Es gibt meistens zwei separate Komponenten, das sind Spezifikation und Body. In Spezifikation wird der Typ Operation deklariert und Body besitzt einen ausführlichen Inhalt der Implementierung. Es versteckt sich für den Benutzer von diesem Paket.
Die Spezifikation hat auch einen sichtbaren und unsichtbaren Teil. Im sichtbaren Teil können alle, z. B Typ, Subprogramme nicht nur in diesem Paket genutzt werden, sondern auch in anderen Paketen. Aber beim unsichtbaren Teil geht das nicht.
Wenn man einen unsichtbaren Teil erzeugen möchte, verwendet man den Befehl ‚Private'. Er kann den sichtbaren Teil zum unsichtbaren Teil deklarieren.

Z. B type Stack is private;

Es gibt auch private Teile in diesem Programm:

```
Type Basic_Vector is abstract tagged limited
   Record
      Vector_Values: Real_Array;
      Error_Value     : Vector_Error_Type:= Not_Initialized;
   end record;
```

2.1.3.3 Record Typen

Record Typen gehören auch zum Composite Type. Ein Record Typ besitzt eine Komponente, der schon ein Name gegeben wurde.

Z. B Subtype Days is Integer range 1…31;

```
Type months is (Jan, Feb, Mar, Apr, May, June, July, Aug, Sept, Oct, Nov, Dec);
Subtype Years is Integer range 1…4000;
Type Date is
  Record
  Day: Days;
  Month: Months;
  Year: Years;
  End Record;
```

Der Subtyp dieser Komponente muss durch einen expliziten Subtyp versorgt werden. Wenn man eine Komponente von Record verwenden möchte, nutzt man das Punktzeichen "."

Z. B. D, E: Date;

```
…
D:=E;                          -- Befehl von ganze Record
D.Years = E.Years+1 ;          -- Befehl von  Year Komponente
```

Es gibt auch einen erweiterbaren Record Type. Er unterscheidet sich vom normalen Record_Type und nutzt Word Tagged in seiner Deklaration.

Z. B.

```
Type Objekt is tagged
        Record
            X_Coord, Y_Coord: Float;
        End Record;
```

Tagged Types sind das Ada/ Spark Konstrukt für die objektorientierte Programmierung.
Diese Tagged Types sind besser für die Deklaration von unterschiedlichen Types in dem geometrischen Objekt.

2.1.3.4 Basic Vector type

In dieser Vektoroperation sollten immer zwei Ergebnisse erzielt werden. Das erste Ergebnis ist ein arithmetisches Ergebnis der Vektoroperation. Und dieses Ergebnis gehört zum Array_Type. Das zweite Ergebnis ist ein Zustand des Vektors. Es zeigt, ob der Wert in Ordnung ist, oder Fehler, wie Überlauffehler, aufgetreten sind. Dieses Ergebnis gehört zum Vector_Error_Type. Deshalb kann nicht einfach der Array_Type für alle Vektoren in dieser Vektoroperation definiert werden. Dann sollte ein Basic_Vector_Type erzeugt werden und dieser besitzt die beiden Typen der Ergebnisse.

```
Type Basic_Vector is abstract tagged limited
   Record
      Vector_Values: Real_Array;
      Error_Value    : Vector_Error_Type:= Not_Initialized;
   end record;
```

Hier kann man sehen, dass der Vector_Values und Error_Value in Basic_Vectors definiert wurde. Und wenn man den Vektor als Basic_Vector definiert, enthält dieser Vektor diese beiden Typen automatisch.

2.2 Vor- und Nachbedingung

2.2.1 Bedeutung der Vor- und Nachbedingung

In Ada 2014 und Spark wurden die Vor- und Nachbedingung für das Programm entworfen. Die entworfene Vorbedingung kontrolliert die eingegebene Variable, bevor das Programm läuft.

Ada 2014 und Spark 2012 besitzen ein neues Format für die Vor- und Nachbedingung. In der alten Version, z. B. Ada95, wurden vor der Bedingung die Zeichen ‚–#' verwendet, um eine Vor- und Nachbedingung definieren zu können.

Betrachte man nun folgendes Unterprogramm [1]

```
Procedure Divide (M, N: in Interger; Q, R: out Interger)
--# derives Q, R from M, N;
--# pre (M>=0) and (N>0);
--# post (M = Q * N + R) and (R < N) and (R >= 0);
```

Wenn man in Ada 2014 und Spark 2012 die gleiche Vor- und Nachbedingung programmieren will, verwendet man den Befehl ‚With', wie im folgenden Unterprogramm dargestellt:

```
Procedure Divide (M, N: in Interger; Q, R: out Interger) with
pre => (M>=0) and (N>0),
post => (M = Q * N + R) and (R < N) and (R >= 0);
```

Die Verifikationsbedingungen sind manchmal nicht praxiswirksam und genau. Das führt dazu, dass das Programm falsch kompiliert wird. Z. B, Y=√X, falls der Bereich von X nicht in der Spezifikation definiert wird, oder wenn der Anwender eine negativ Anzahl eingegeben hat, kann die Funktion vielleicht nicht ausführt werden. Deshalb benutzte man die Vor- und Nachbedingung, um die eingegebenen Werte vor dem Lauf des Programms zu testen.
Die Vorbedingung sagt aus, welche Voraussetzungen die Funktion erfüllen muss. Wenn sie gilt, so müssen nach Ausführung der Funktion alle Nachbedingungen erfüllt sein, sonst ist das Programm nicht korrekt.
Die Nachbedingungen einer Funktion oder eines Programms machen deutlich, welche Aussagen nach der Ausführung gelten müssen, falls zuvor die Vorbedingungen erfüllt waren.

Betrachte man nun die Unterprogramm-Spezifikation:

```
Procedure WurzelQuadrat(M: in Float; Q: out Float) with
pre => (M>=0),
post=>  (Q = sqrt(M));
```

Die Vorbedingung bedeutet, dass die Prozedur nur implementiert wird, wenn die eingegebenen Werte M nicht negativ sind.
Betrachte man nun die obere Unterprogramm-Spezifikation. Die Nachbedingung bedeutet, dass die Ausgabe eine mathematische Beziehung bei der Quadratwurzel haben muss.

2.2.2 Wichtigkeit der Vor- und Nachbedingung

In Spark darf man für eine Prozedur oder Funktion die Vor- und Nachbedingung definieren. Dann stellt sich die Frage, in Spark werden alle Fehler in der Laufzeit automatisch kontrolliert, aber warum braucht man auch eine Vor- und Nachbedingung. In Spark spielt die Vor- und

Nachbedingung eine sehr wichtige Rolle. Nachfolgend soll nun die Wichtigkeit der Vorbedingung erläutert werden.

Die erste Funktion der Vorbedingung kann die Zerstörung des Programms vermeiden. Durch die definierte Vor- und Nachbedingung wird der eingegebene Wert kontrolliert, bevor das Programm gelaufen ist. Es kann die Fehler in der Compilezeit prüfen.

Die zweite Funktion der Vorbedingung ist die Vereinfachung des Tests. Mit Hilft der Vorbedingung kann man viele Testschritte verkleinern und einsparen. Zum Beispiel bei der Addition von zwei Vektoren. Jeder Vektor besitzt 1000 Elemente. Wenn der Befehl ausgeführt wird, müssen alle Summen von 1000 Vektoren kontrolliert werden. Es ist zu kompliziert. Aber wenn man eine Vorbedingung an den Anfang setzt, zum Beispiel zwei Vektoren Left(Index) und Right(Index), wenn der Wert von Left oder Right größer als der halbierte Maximalwert ist, dann kann man die Addition dieser Reihe kontrollieren. Das bedeutet aber auch, wenn der Wert der beiden Vektoren kleiner als der halbierte Maximalwert ist, kann die Kontrolle nicht ausgeführt werden. In der Compilezeit wird nur der eingegebene Wert, der von der Vorbedingung ausgewählt wurde, geprüft.

Die Nachbedingung ist ein Kontrollsystem, mit dem man überprüfen kann, ob die eingegebenen Werte nach der Prozedur zu einem richtigen Ergebnis geführt haben.

3 Erarbeitung des Lösungskonzeptes-Entwicklung der Bibliothek

In zweitem Teil werden die Aufgaben gestellt und Lösungskonzepte entwickelt. Gemäß dem Lösungskonzept werden nun die Arbeiten durchgeführt. Die Erarbeitung besteht drei Teilen, die Analysierung und Veränderung der Struktur der Bibliothek, die Analysierung von Vektoroperationen und der Entwurf der Hilfsfunktionen der Vorbedingung und Nachbedingung, dann die Verbindung aller Bedingungen für jede Vektoroperation. Um die Bibliothek besser entwickeln zu können, musste man sich am Anfang zunächst die Grundlagen des Softwareentwurfs aneignen.

Vor der Entwicklung des Softwaresystems muss man normalweise die Software entwerfen. Um eine bessere Bibliothek entwickeln zu können, sollte man den Systementwurf kennen. Anhand (Ian Sommerville, 2001) kann man sehen:

Der Systementwurf beschäftigt sich damit, wie die Funktionalität des Systems von den verschiedenen Komponenten zur Verfügung gestellt wird. Die Aktivitäten in diesem Prozess sind:

1. Aufteilung der Anforderung: Die Anforderungen werden analysiert und in zusammenhängende Gruppen aufgeteilt: Es gibt normalerweise verschiedene Möglichkeiten der Aufteilung, so dass zu diesem Zeitpunkt mehrere Alternativen erstellt werden können.

2. Bestimmung des Subsystems: In dieser Phase werden die verschiedenen Subsysteme bestimmt, die einzeln oder zusammen die Anforderungen erfüllen können. Gruppen von Anforderungen stehen oft in Verbindung zu Subsystemen, so dass die Schnittstellendefinition und die Aufteilung der Anforderung zu einer Aktivität zusammengefasst werden können. Die Bestimmung des Subsystems kann aber auch durch andere organisatorische oder umgebungsbedingte Faktoren beeinflusst werden. [2]

Der Softwareentwurf gehört zu den Hauptaktivitäten der Systementwicklung. Nach erfolgter Problemanalyse und Anforderungsdefinition wird die Grundlage für die Software-Implementierung geschaffen.

3.1 Die Struktur der Bibliothek

Der erste wichtige Schritt bei der Programmierung ist, dass ein allgemeiner Programmentwurf erstellt wird. Eine vorherige Analyse kann zu besseren Lösungen führen. Bei allen Vorgehensweisen ist es wichtig, den Softwareentwurf mit den heute zur Verfügung stehenden rechnergestützten Werkzeugen mit der richtigen Analyse und dem richtigen Design durchzuführen.
Im dritten Teil wurde das Prinzip des Systementwurfs erkannt. Durch dieses Prinzip wurde die Bibliothek in zwei verschiedene Aufgaben aufgeteilt. Die Bibliothek heißt Test-Driver. Sie besteht aus zwei Projekten, Aunit' und, Test-Vectoren'. Projekt, Aunit' ist ein Satz von Ada-Paketen auf der Grundlage der Unit Familie von Unit-Test-Frameworks. Es kann als Werkzeug eines Entwicklers das Schreiben und die Entwicklung der Ada-Software erleichtern. Eines der Hauptziele ist es, Unit-Tests einfach zu entwickeln und auszuführen, aber nicht für das Prozessmanagement zu machen. Das Framework unterstützt die einfache Zusammensetzung der Gruppen von Unit-Tests bei der Bestimmung, welche Tests für einen bestimmten Zweck laufen und Flexibilität bieten. In dieser Arbeit brauchte man das Projekt Aunit nicht zu entwickeln. Die Entwicklung des Projektes, Test-Vectoren' war das Hauptziel. Das Projekt besteht aus drei Teilen, Test, Aunit und Vectoren. Das Projekt Vectoren verwirklicht die Operation der Vektoren. Andere adb Dateien wie Vectoren-aux.adb und Vectoren-Standard_Vectoren sind die Kindpakate von Vectoren.adb. Sie besitzen die Implementierung der Hilfsfunktion und erben alle Typen von Vector.adb. Eine eigene wichtige Funktion wird im Unterprogramm geschrieben, z. B. eine Funktion zur Prüfung des Ergebnisses oder eine Funktion zur Eingabe oder Ausgabe der Werte von Vektoren.
Eine Aufgabe bestand auch darin, Vor- und Nachbedingungen für jede Vektoroperation zu entwerfen. Im Verlauf dieses Studiums wurde bereits die Grundstruktur für jede Vektoroperation programmiert, aber für diese neue Aufgabe sollte die Grundstruktur verändert werden. Um Fehler zu vermeiden, wurden alle mathematischen Vektoroperationen im Vector.ops eingesetzt und alle Hilfsfunktionen in Vector.cond programmiert.
Nach der Programmierung von Vectoren.adb sollte ein Prüfprogramm entworfen werden, um die Richtigkeit des Programms zu bestimmen und alle Fehlerzustände jeder Prozedur zu testen. In Vectoren-basic_vector_test_data-bassic_vector_tests.adb. In dieser Datei sollten alle möglichen Fehler analysiert und programmiert werden, die nicht nur bei jeder Prozedur, sondern auch bei allen Vor- und Nachbedingungen in Vectoren.ops.adb auftreten können. Im Testteil konnte die Robustheit und Flexibilität des Programms erhöht werden. Die Struktur der Bibliothek kann in einem Paketdiagramm dargestellt werden:

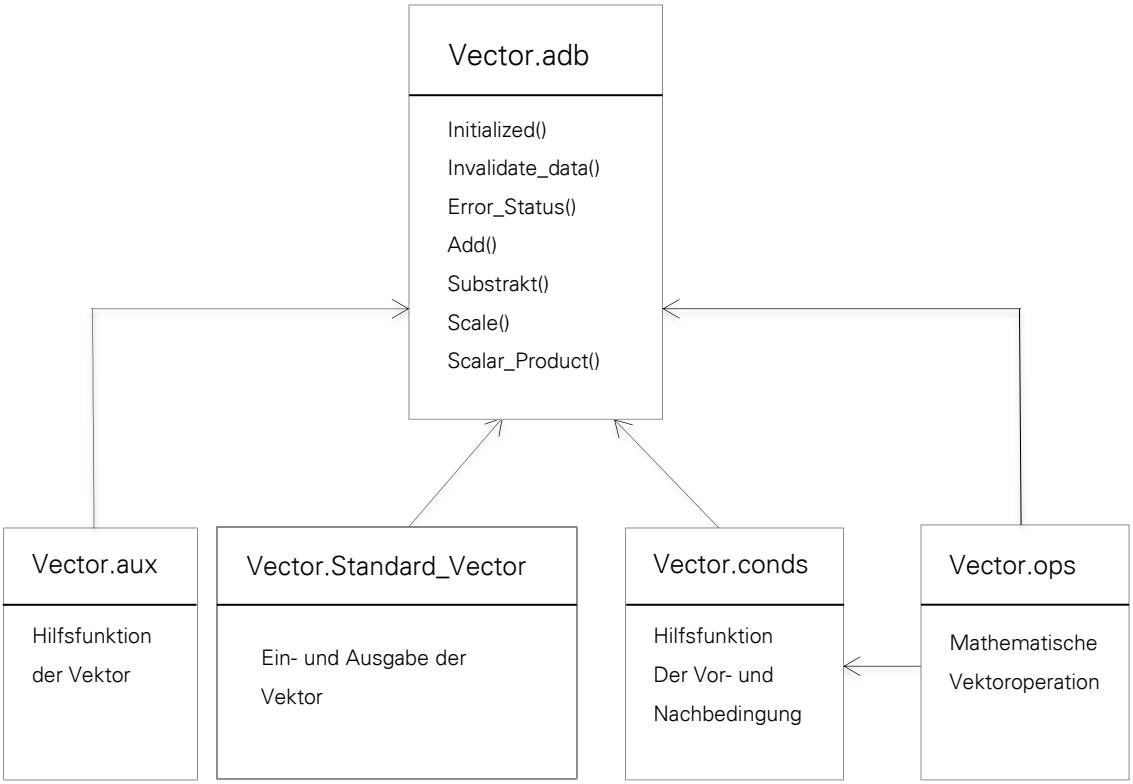

Abbildung 2: 3.1 Struktur der Vektoren

In jedem Projekt sind die Dateien .adb die Implementationsdateien. Die Dateien .ads sind die Deklarationen. Wenn eine Implementationsdatei die Funktionen von anderen Dateien benutzen möchte, muss sie die Deklarationen ‚with' oder ‚use' hinzufügen, die am Anfang der Datei liegt. Auch bei der Programmierung wurden die Standardbibliotheken ‚Ada.Text_IO' und ‚GNATtest_Generated' zum Test verwendet. Die Struktur des Testprogramms wird in nachfolgender Tabelle vorgestellt.

Test	
Vectoren-basic_vector_test_data-bassic_vector_tests.adb Vectoren-basic_vector_test_data-bassic_vector_tests_aux.adb Vectoren-filter_parameters-filter_parameters_test_data-filter_parameters_tests.adb Vectoren-filter-parameters-filter_parameters_test_data.adb Vectoren-standard_Vectoren-real_vector_test_data-real_vector_tests.adb Vectoren-standard_Vectoren-real_vector_test_data.adb Vectoren-time_series-time_series_test_data-time_series_tests.adb Vectoren-time_series-time_series_test_data.adb	Vectoren-basic_vector_test_data-bassic_vector_tests.ads Vectoren-basic_vector_test_data-bassic_vector_tests_aux.ads Vectoren-basic_vector_test_data.ads Vectoren-filter_parameters-filter_parameters_test_data-filter_parameters_tests.ads Vectoren-filter_parameters-filter_parameters_test_data.ads Vectoren-filter_parameters-test_data-tests.ads Vectoren-filter_parameters-test_data.ads Vectoren-standard_Vectoren-real_vector_test_data-real_vector_tests.ads Vectoren-standard_Vectoren-real_vector_test_data.ads Vectoren-standard_Vectoren-test_data-tests.ads Vectoren-standard_Vectoren-test_data.ads Vectoren-test_data.ads Vectoren-time_series-test_data-tests.ads Vectoren-time_series-test_data.ads Vectoren-time_series-time_series_test_data-time_series_tests.ads Vectoren-time_series-time_series_test_data.ads

Tabelle 1: 3.1 Struktur der Bibliothek 2

3.2 Bearbeitung der Struktur von Vektoroperationen

Zuerst wurde der Typ, Basic_Vector Typ in Vectors.ads definiert.

```
Type Basic_Vector is abstract tagged limited
    Record
       Vector_Values: Real_Array;
       Error_Value    : Vector_Error_Type:= Not_Initialized;
   end record;
```

Dieser Basic_Vector Typ besitzt nicht nur ein arithmetisches Ergebnis der Vektoroperationen (Array_Typ), sondern auch den Zustand der Vektoren (Vektor_Error_Typ). Das ist ein gründlicher Typ unserer Vektoroperationen. Ein Basic_Vektor muss gleichzeitig die Werte und den Zustand besitzen.
Die drei Zustände sind, Not_Initialized, No_Error und Overflow_Error.
Nachfolgend werden diese drei Zustände näher erläutert.

Not_Initialized:	**Die Werte von Vektoren werden nicht initialisiert. Der Wert im Vektor setzt sich nicht auf 0.0.**
No_Error:	**Vektoren haben keine Fehler. Es ist ein richtiger Zustand.**
Overflow_Error:	**Die Werte von Vektoren überschreiten die Grenze. Dies passiert nach der Berechnung der Vektoren.**

In diesem Praktikum wurde bereits Vectors.adb geschrieben, diese Vectors.adb besitzt alle Vektoroperationen, die nachfolgend aufgeführt sind:

- Initialize, setzen alle Elemente im Vector auf 0.0, setzen ‚Error_Value' im Vector zum ‚No_Error'.
- Invalidate_Data, setzen Fehler zum ‚Not_Initialized', was bedeutet, dass die existierten Werte keine Bedeutungslosigkeit haben.
- Error_Status, bekommt den ‚error status', um die Prüfung von Resultaten und Dateien zuzulassen
- Add, nimmt Vectoren ‚Left' und ‚Right' und reserviert die Gesamtsumme im ‚Result'.
  Add gehört zur mathematischen Operation
- Subtract, nimmt Vectoren ‚Left' und ‚Right' und reserviert den Rest im ‚Result'.
  Subtract gehört zur mathematischen Operation.

- Scale, nimmt Vector ‚Left' und Factor und reserviert das Produkt von ‚Left' und Factor im ‚Result', gehört auch zur mathematischen Operation
- Scalar_Product, nimmt Werte von ‚Left' und ‚Right' und rechnet ‚Scalar_Product'. Wenn der Wert überschritten wird, oder irgendein zwischen ‚Left' und ‚Right' Fehler habt, will ‚Error_Status' ‚Overflow_Error ' zurückgeben und es gehört zur mathematischen Operation

Am Anfang wurde versucht, durch den mathematischen Beweis unterschiedliche Funktionen in Vector_aux.adb zu programmieren. Dann könnte man im Vectors.ads die Funktion als Vor- und Nachbedingung aufrufen und kombinieren. Aber es ist nicht möglich wegen eines Fehlers, Circular Unit Dependency. Nachfolgend wird erklärt, was diese Fehler bedeuten und warum die Grundstruktur der Vectors.adb verändert werden muss.

Circular Unit Dependency

Diese Fehler passieren, wenn zwei Unterprogramme voneinander abhängig sind.
Z. B. wurden zwei Pakete programmiert:

```
A.adb                              B.adb
With B                             With A
Prozedur A( in : X   ; Out: Y  );  Prozedur B( in : X   ; Out: Y  );
Inhalt der Implementierung         Inhalt der Implementierung
End A;                             End B;
```

Dann kann man sehen, A.adb verwendet B, um A zu erzeugen, aber gleichzeitig verwendet B auch A, um B zu erzeugen. Das darf nicht in Ada oder Spark passieren, und dann wurde ein Fehler Circular Unit Dependency erzeugt.
Wenn dieser Fehler auftritt, kann man ihn einfach durch das Kindpaket beseitigen.
Zum Beispiel wurde das Kindpaket von Prozedur A programmiert.
Das erste Paket A ist:

```
A.adb
```

Dann wurde das Kindpaket für A programmiert:

```
A.Ops.adb( Kindpaket der A )
With B
Prozedur A( in : X   ; Out: Y  );
Inhalt der Implementierung
End A;
```

Danach wurde das Paket B programmiert:

```
B.adb
With A
Prozedur B( in : X   ; Out: Y  );
Inhalt der Implementierung
End B;
```

Die Beziehung siehe folgende Abbildung.

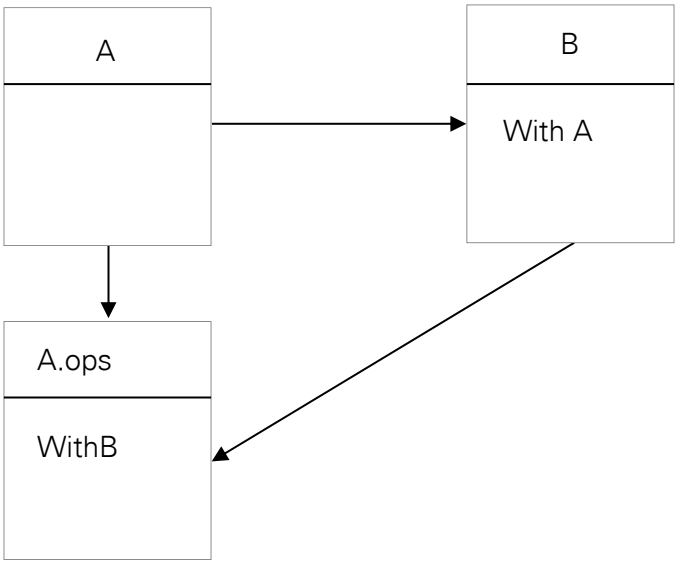

Abbildung 3: 3.2.1 Unit Circular Dependency

Das Kinpaket A verwendet B und B verwendet A. Dann tritt dieser Fehler nicht mehr auf.

In diesem Programm wurden alle Vor- und Nachbedingungen im Vector.aux programmiert. Aber Vector.aux ist schon ein Kindpaket. Es gibt schon eine Verbindung zwischen diesen zwei Dateien. Wenn man im Vector.ads die Hilfsfunktion von Vektor.aux verwendet, ist es nicht möglich.

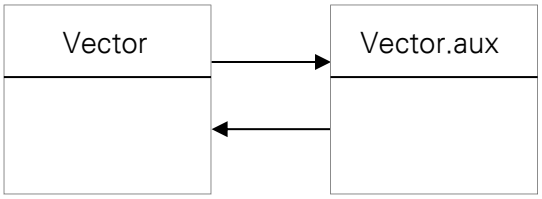

Abbildung 4: 3.2.2 Unit Circular Dependency

Dann wurde die Grundstruktur des alten Vector.ads verändert. Es wurden zwei neue Adb Dateien erzeugt, sie sind Vector.cond und Vector.ops. Alle mathematischen Vektoroperationen im Vektor.adb (Add, Substrakt, Scale, Scalar_Product) wurden in den Vector.ops hineingelegt und im Vektor.conds wurden alle Vorbedingungen und Nachbedingungen als Hilfsfunktion programmiert. Die Beziehung siehe Abbildung.

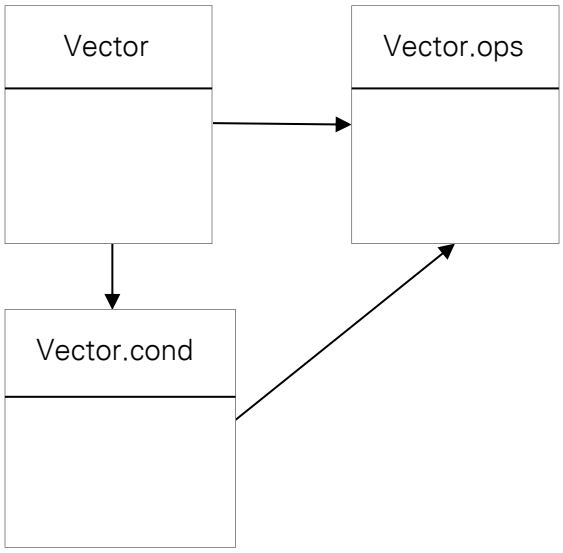

Abbildung 5: 3.2.3 Unit Circular Depency

Danach konnte man erkennen, Vector.ops und Vector.conds sind beides Kindpakate des Vektors. Die Vor- und Nachbedingung im Vector.ops wurde programmiert und nur die Hilfsfunktion im Vectors.conds verwendet. So wurde eine richtige Struktur aufgebaut.

3.3 Programmierung der Vor und Nachbedingung

Wenn man die Vor- und Nachbedingung für die Vektoroperation entwerfen möchte, muss man zuerst mit Hilfe eines mathematischen Beweises sicherstellen, dass alle Vorbedingungen mathematisch richtig sind. Der Teil der mathematischen Beweise ist in der Bachelorarbeit von Herrn Zhang zu finden. Dort wurde die Richtigkeit aller Vorbedingungen für jede Vektoroperation durch mathematische Beweise nachgewiesen. Deshalb wird nun direkt der Programmanteil beschrieben. Um alle Vor- und Nachbedingungen für die Programmierung fehlerfrei und sauber entwickeln zu können, werden nun alle Schritte der Bedingungen der Hilfsfunktion entworfen. Z. B. wurden A, B zwei Hilfsfunktionen der Vorbedingung der Prozedur X definiert.

Dann, wenn man Vor- und Nachbedingung mit X definiert, folgt:

```
Prozedur X ( ) with
Pre => (A and then B);
```

Es sieht einfach und sauber aus und man kann auch Fehler vermeiden und alle Hilfsfunktionen im Vector.conds wurden programmiert. Anschließend werden der Code und das Prinzip der Hilfsfunktion in Vector.conds näher erläutert.

3.3.1 Scale

Scale in den Vektoroperationen bedeutet eine Multiplikation zwischen einem Vektor und einem Faktor, deshalb, wenn man die Vor- und Nachbedingung für Scale programmiert, ist es ein bisschen einfach. Deshalb wird die Multiplikation zu Beginn erklärt.

3.3.1.1 Vorbedingung von Scale

Für Scale gibt es nur einfache und komplexe Vorbedingungen, weil der mathematische Vorgang relativ einfach ist. Weil der Faktor bestimmt ist, sollte man nur an den Vektor denken. Bei einer einfachen Vorbedingung sollte man nur den Maximalwert des Vektors finden. Wenn das Ergebnis der Multiplikation zwischen dem Maximalwert mit dem Faktor nicht größer als die Grenze ist, laufen die Ergebnisse der anderen Werte des Vektors nicht über. Wenn die einfache Vorbedingung nicht erfüllt ist, müssen alle Elemente im Vektor geprüft werden.
Wenn die Multiplikation ausgeführt wird, entsteht ein Überlauffehler, weil das Ergebnis einen zu hohen Wert bekommen hat. Aber denken Sie an die Vorbedingung, die entworfen wurde. Der eingegebene Wert wird zuerst durch die Vorbedingung kontrolliert, bevor das Programm läuft. Deshalb kann der Überlauffehler am Ende durch die Verwendung der Vorbedingung vermieden werden.
Um diese einfache Vorbedingung realisieren zu können, wird die Hilfsfunktion , AreValueScalable' programmiert.

```
function AreValueScalable(Left, Right : in Real_Type) return Boolean is
Result : Boolean;
 begin
      Result :=True;
      for Index in Vector_Index loop
          Result:= Result and Aux.Is_Scalable(Left  => Left, Right => Right);
     end loop;
    return Result;
 end AreValueScalable;
```

Weil der Faktor und Maximalwert im Vektor beide einen Wert darstellen, wurden am Anfang zwei Real_Type Werte eingegeben. Dann ist durch die Hilfsfunktion Aux.Is_Scalable zu bestimmen, ob die beiden Werte multipliziert werden.
Die Komplexe Vorbedingung ist ähnlich wie andere Vektoroperationen, alle Werte der Vektoren müssen mit dem Faktor multipliziert und alle Ergebnisse geprüft werden. Dann wurde die Vorbedingung der Scala realisiert.

```
procedure Scale (Result : out Basic_Vector; Left   : in Basic_Vector; Factor : in Real_Type)
  with
      pre => conds.AreValueScalable(Left  => Left.Max_Norm,Right => Factor)
               and then
                        (for all Index in Vector_Index =>
                        (Aux.Is_Scalable(Left.Vector_Values(Index), Factor )));
```

Der obige Code ist die Vorbedingung von Scale. Wenn die einfache Bedingung nicht erfüllt wird, geht es in den komplexen Vorbedingungsteil.

3.3.1.2 Nachbedingung von Scale

Die komplexe Vorbedingung ist ähnlich wie andere Vektoroperationen. Man beachte die mathematische Beziehung zwischen dem Ergebnis und der Operation. Alle Ergebnisse müssen kleiner als die Grenze sein. Dann wurde wie folgt programmiert:

```
post => (for all Index in Vector_Index =>
          (Result.Vector_Values(Index) = Left.Vector_Values(Index)* Factor))
              and then
                        Conds.ResultinrangeScala1(Left  => Left,
                                                  Right => Factor);
```

Der obige Code zeigt, dass man in der Nachbedingung von Scale auch die mathematische Beziehung zwischen Ergebnis und eingegebenem Vektor und Koeffizient analysieren muss. Wenn Scala richtig ausgeführt wurde, dann war die Gleichung Result= Left* Faktor erfüllt. Dann musste getestet werden, ob alle erzielten Ergebnisse innerhalb der Grenze sind.

3.3.2 Add

Beim mathematischen Beweis wurden drei Vorbedingungen und zwei Nachbedingungen für die Addition zwischen zwei Vektoren beschrieben. Hier wurde nicht nur eine Vorbedingung entworfen, weil der Vorgang der Addition komplexer als der Vorgang der Multiplikation ist. Anschließend werden diese fünf Bedingungen in der Addition ausführlich erläutert.

3.3.2.1 ArevalueAddable (Einfache Vorbedingung)

Die erste Vorbedingung von Add lautet, die Summe der Maximalwerte in zwei Vektoren müssen kleiner als die Grenze, Real_Type'Last sein.

Die Grundlage dieses Schrittes ist, wenn die Summe der Maximalwerte der beiden Vektoren kleiner als die Grenze ist, bedeutet die Summe aller Elemente kleiner als die Grenze. Dann kann die Berechnung richtig ausgeführt werden. Wenn die Summe der Maximalenorm größer als die Grenze ist, weiß man nicht, ob die Summe der beiden Vektoren in Ordnung ist, oder überläuft. Deshalb braucht man nicht nur eine einfache Vorbedingung, sondern auch später eine mittlere und komplexe Vorbedingung. In diesem Schritt, wenn die Summe der zwei Vektoren überläuft, zeigt das Ergebnis einen Überlauffehler und das Boolesche Ergebnis am Ende einen ‚Nein' Zustand an.

Die erste Hilfsfunktion wird als AreValueAddable bezeichnet. Man gibt die Maximalnorm von zwei Vektoren ein und dann entscheidet die Funktion automatisch, ob die Summe in der Grenze bleibt und zum Schluss wird ein boolesches Ergebnis ausgegeben.

Der programmierte Code lautet:

```
function AreValueAddable(Left, Right : in Real_Type)   return Boolean is
Result : Boolean;
    begin
        Result :=True;
        for Index in Vector_Index loop
        Result:= Result and Aux.Is_Addable(Left  => Left,
                                           Right => Right);
   end loop;
return Result;
```

Am Anfang wurden zwei Real_Type Werte zur Funktion eingegeben. Sie sind die Maximalnorm der beiden Vektoren. Die Maximalnorm bedeutet den Maximalwert im Vektor, deshalb gehört er zum Real_Type, aber nicht zum Array_Type. Dann wurden die zwei Maximalwerte der Vektoren durch die Hilfsfunktion Aux.Is_Addable kontrolliert, ob die beiden Werte addiert werden können. Zum Schluss wird ein boolesches Ergebnis ausgegeben. Wenn das Ergebnis am Ende ein ‚Ja' bekommt, bedeutet es, dass das Programm richtig läuft. Sonst wird der Wert der mittleren Vorbedingung geprüft. Es ist die einfache Vorbedingung der Addition, weil nur ein Wert von beiden Vektoren kontrolliert wird.
Wenn diese Vorbedingung erfüllt wurde, kann in Vector.ops.ads schon diese Hilfsfunktion verwendet und die Vorbedingung einfach programmiert werden.
Der Code lautet:

```
procedure Add (Result      : out Basic_Vector;
               Left, Right : in Basic_Vector)
  with
            pre => conds.AreValueAddable(Left  => Left.Max_Norm,
                                         Right => Right.Max_Norm);
```

Bei der Programmierung der Vorbedingung wird direkt die Hilfsfunktion aufgerufen und dann die Parameter in die Hilfsfunktion eingegeben. Dann wird die Hilfsfunktion aufgerufen und ausgeführt. Mit dem zurückgegebenen booleschen Ergebnis entscheidet das Programm automatisch, ob die Funktion Add die Vorbedingung erfüllt.

3.3.2.2 PreAdd (Mittlere Vorbedingung)

Wenn die einfache Vorbedingung nicht erfüllt wurde, kann nicht bestimmt werden, ob die Summe der beiden Vektoren in Ordnung ist, oder überläuft. Deshalb braucht man noch eine mittlere Vorbedingung, um den eingegebenen Wert weiter zu prüfen.
Für die Programmierung der Vor- und Nachbedingung braucht man nicht nur eine einfache Vorbedingung, sondern auch eine mittlere und komplexe Vorbedingung. Das funktioniert, nachdem die einfache Vorbedingung ausgeführt wurde. Die mittlere Vorbedingung ist komplexer als die einfachere Vorbedingung, aber sie ist exakter. Mit Hilfe der mittleren Vorbedingung kann man auch den Prüfschritt sparen und sauberer machen.
Die Grundlage der mittleren Vorbedingung in Add wurde bereits in den mathematischen Grundlagen erklärt. Wenn der Wert der Reihe im Vektor größer als Real_Type'Last / 2 (halb der Grenze) ist, ergibt sich die Notwendigkeit, die Addition der Reihe zu kontrollieren.

Der Code für die mittlere Vorbedingung der Addition lautet:

```
function PreAdd(Left, Right : in Basic_Vector) return Boolean is
Result : Boolean;
LeftV, RightV : Real_Type;
begin
        Result :=True;
                for Index in Vector_Index loop
                LeftV := Left.Vector_Values (Index); RightV := Right.Vector_Values (Index);
                        If  (Abs(LeftV)>= Real_Type'Last/2.0) or (Abs(RightV)>=Real_Type'Last/2.0)   then
                                Result:= Result and Aux.Is_Addable(Left  => LeftV,Right => RightV);
                        end if;
                end loop;
        return Result;
end PreAdd;
```

Anhand des obigen Codes kann man ersehen, dass zuerst zwei Vektoren eingegeben wurden, dann wurden die Werte der Vektoren kontrolliert. Wenn der Wert einer Reihe größer als Real_Type'Last ist, wird die Aux.Is_Addable aufgerufen und bestimmt, ob die Werte dieser Reihe addiert werden können. Am Ende wird auch ein boolesches Ergebnis ausgegeben.
Im Vector.ops muss auch diese mittlere Vorbedingung ergänzt werden:

```
procedure Add (Result  : out Basic_Vector;  Left, Right : in Basic_Vector)
 with
 pre =>conds.AreValueAddable(Left  => Left.Max_Norm, Right => Right.Max_Norm)
              or else
                      conds.PreAdd(Left  => Left, Right => Right) ;
```

Über die Beziehung zwischen zwei Bedingungen muss man sich noch Gedanken machen. Es gibt zwei Verbindungsmethoden zwischen zwei Bedingungen, sie sind , and then' und ‚or else'. Wenn die erste Bedingung erfüllt wird, dann geht es zu der zweiten Bedingung, man verwendet ‚and then ‚. Wenn die erste Bedingung nicht erfüllt wird, dann geht es zum nächsten Schritt, und man verwendet ‚or else':
Die obigen Codes haben die Beziehung zwei AreValueAddable und PreAdd angezeigt. Wenn AreValueAddable nicht erfüllt wird, dann geht es zum Schritt, PreAdd'.

3.3.2.3 Komplexe Vorbedingung

Wenn die einfache Vorbedingung und mittlere Vorbedingung ausgeführt wurden, aber nicht bestanden haben, muss eine komplexe Vorbedingung entworfen werden. Eine komplexe Vorbedingung kontrolliert alle Elemente, und der Prüfschritt kann nicht eingespart werden. Aber es ist auch sicher, wenn alle Elemente geprüft werden, kann man wirklich die Fehler oder Zerstörung der Prozedur minimieren und vermeiden.
Der Code, die Verbindung aller Vorbedingungen der Addition, ist:

```
procedure Add (Result: out Basic_Vector; Left, Right : in Basic_Vector)
  with
      pre =>  conds.AreValueAddable(Left  => Left.Max_Norm, Right => Right.Max_Norm)
              or else
                    conds.PreAdd(Left  => Left, Right => Right)
                          or else
                          (for all Index in Vector_Index =>
                          (Aux.Is_Addable(Left.Vector_Values(Index), Right.Vector_Values(Index)))),
```

Durch den obigen Code kann man feststellen, dass, wenn die einfachen und mittleren Vorbedingungen beide nicht ausgeführt wurden, wird die komplexe Vorbedingung ausgeführt. Durch die komplexe Vorbedingung werden alle Werte der Vektoren kontrolliert, ob sie addiert werden können. Dieser sichere aber komplexe Schritt spielt am Ende eine wichtige Rolle. Und die Vorbedingung der Addition ist fertig.

3.3.2.4 Nachbedingung der Addition

In der Nachbedingung der Addition sollte man über die mathematische Beziehung nachdenken. Das erzielte Ergebnis muss gleich wie die Summe der beiden Vektoren sein. Dann kann folgendes programmiert werden:

```
post =>  (for all Index in Vector_Index =>
         (Result.Vector_Values(Index) = Left.Vector_Values(Index)+ Right.Vector_Values(Index))) ;
```

Der obige Code macht deutlich, wenn das erzielte Ergebnis gleich wie die Summe der beiden Vektoren ist, wurde die Nachbedingung richtig ausgefüllt.
Bei der Nachbedingung wurde die Hilfsfunktion, ResultinrangeAdd‘ verwendet. Diese Hilfsfunktion kontrolliert, ob sich alle Werte im Ergebnis innerhalb der Grenze befinden.

Dann kann ergänzt werden:

```
post =>  (for all Index in Vector_Index =>
          (Result.Vector_Values(Index) = Left.Vector_Values(Index)+ Right.Vector_Values(Index)))
              and  then
                        Conds.ResultinrangeAdd(Left  => Left, Right => Right);
```

Die zwei Nachbedingungen verbindet man mit, and then'. Das bedeutet, diese beiden Nachbedingungen müssen beide erfüllt sein, dann ist die Nachbedingung bestanden. Dann sind alle Vorbedingungen und Nachbedingungen der Addition der Vektoren fertiggestellt.

3.3.3 Subtraktion

Beim mathematischen Beweis der Substraktion wurden auch vier Vorbedingungen und zwei Nachbedingungen für die Addition zwischen zwei Vektoren beschrieben. Der Vorgang der Subtraktion ist ähnlich wie der der Addition, aber nicht gleich, sondern komplexer. Anschließend werden ausführlich diese sechs Bedingungen in der Subtraktion erläutert.

3.3.3.1 AreValueSubstraktable (Einfache Vorbedingung)

Bei der Addition müssen nur die Maximalwerte der zwei Vektoren berücksichtigt werden, aber die Subtraktion unterscheidet sich von der Addition darin, dass man über die Maximalwerte und Minimalwerte beider Vektoren nachdenken muss. Die erste, einfache Vorbedingung wurde als, AreValueSubstraktable' Hilfsfunktion programmiert.
Die mathematische Grundlage dieses Schrittes ist die, wenn das Ergebnis der Subtraktion zwischen Maximalwert und Minimalwert in Ordnung ist, bedeutet das, dass alle Ergebnisse nach der Berechnung in Ordnung sind. Wenn das Ergebnis überläuft, weiß man nicht, ob es nach der Subtraktion in Ordnung ist oder überläuft. Deshalb braucht man hier auch die mittlere und komplexe Vorbedingung wie die Addition.
In diese Funktion wird der Maximalwert und Minimalwert von den beiden Vektoren eingegeben und es wird die Substraktion zwischen dem Maximalwert von einem Vektor und der Minimalwert von dem anderen Vektor durch die Hilfsfunktion Aux_Is.Substraktable analysiert, ob diese beiden Werte subtrahiert werden können.

Dann wird folgender Code programmiert:

```
-------------------------------------------------------------------------------------------------------
-- AreValueSubstraktable (Max-Min), Einfache Vorbedingung der Substraktion-------
-------------------------------------------------------------------------------------------------------
function AreValueSubstraktable1(Left, Right : in Basic_Vector) return Boolean is
      Result : Boolean;
      RightMin : Real_Type;
           begin
           Result :=True;
           RightMin := Vectors.Min(Left => Right);
                  for Index in Vector_Index loop
                      Result:= Result and Aux.Is_Subtractable(Left  => Left.Max,
                                                              Right => RightMin) ;
                  end loop;
     return Result;
end AreValueSubstraktable1;
-------------------------------------------------------------------------------------------------------
-- AreValueSubstraktable(Min-Max), Einfache Vorbedingung der Substraktion-------
-------------------------------------------------------------------------------------------------------
function AreValueSubstraktable2(Left, Right : in Basic_Vector) return Boolean is
     Result : Boolean;
     LeftMin : Real_Type;
          begin
             Result :=True;
             LeftMin := Vectors.Min(Left => Left);
              for Index in Vector_Index loop
                    Result:= Result and Aux.Is_Subtractable(Left  => LeftMin,
                                                            Right => Right.Max) ;
            end loop;
     return Result;
end AreValueSubstraktable2;
```

Maximalwert und Minimalwert sind ein Wert im Vektor, deshalb gehören sie zu Real_Type, aber nicht zum Array_Type. Am Anfang wurde der Wert eingegeben, dann wurden die Werte mit Hilfe der Funktion Aux.Is_Substrakt überprüft, ob sie in Ordnung sind.

Im Vectors.ops.ads kann schon die Vorbedingung der Substraktion programmiert werden:

```
procedure Subtract (Result      : out Basic_Vector;
                    Left, Right : in Basic_Vector)
  with
  Pre => (conds.AreValueSubstraktable1(Left  => Left, Right => Right)
          and then
              conds.AreValueSubstraktable2(Left  => Left, Right => Right))
```

Der obige Code zeigt, dass das Ergebnis der Substraktion zwischen dem Maximalwert der Left und Minimalwert der Right und die Subtraktion zwischen dem Maximalwert der Right und Minimalwert der Left geprüft und erfüllt werden muss. Dann ist die einfache Vorbedingung der Substraktion fertig.

3.3.3.2 PreSubstrakt (Mittlere Vorbedingung)

Wenn die einfache Vorbedingung nicht erfüllt ist, weiß man nicht, ob das Ergebnis am Ende in Ordnung ist oder überläuft. Deshalb wird der eingegebene Wert auch durch die mittlere Vorbedingung geprüft. Die mittlere Vorbedingung wurde als PreSubstrakt programmiert und die Grundlage wurde bereits im Beweis erklärt. Wenn der Absolutwert von einer Reihe der Vektoren größer als Real_Type'Last (halb der Grenze) ist, ergibt sich die Notwendigkeit, die Werte von dieser Reihe zu berechnen und das Ergebnis zu kontrollieren.

```
function PreSubtract (Left, Right : in Basic_Vector) return Boolean is
Result : Boolean; LeftV, RightV : Real_Type;
    begin
        Result :=True;
            for Index in Vector_Index loop
                LeftV := Left.Vector_Values (Index);RightV := Right.Vector_Values (Index);
                    If (Abs(LeftV)>= Real_Type'Last/2.0) or (Abs(RightV)>=Real_Type'Last/2.0)
                        Then Result:= Result and Aux.Is_Subtractable(Left  => LeftV,
                                                                     Right => RightV);
                end if;
            end loop;
      return Result;
 end PreSubtract;
```

Der obige Code gehört zur mittleren Vorbedingung der Subtraktion. Am Anfang wurden zwei Vektoren eingegeben, und dann, wenn der Wert eines der Vektoren größer als Real_Type'Last/2.0 ist, werden die Werte beider Vektoren durch die Hilfsfunktion Aux_Is_Substrakt überprüft, ob diese Werte subtrahiert werden können.
Dann wird die mittlere Vorbedingung mit der einfachen Vorbedingung programmiert:

```
Pre => (conds.AreValueSubstraktable1(Left  => Left, Right => Right)
        and then
                conds.AreValueSubstraktable2(Left  => Left, Right => Right))
                        or else
                                conds.PreSubtract(Left  => Left, Right => Right);
```

Im Beweis wurde erklärt, dass, wenn die erste Vorbedingung nicht erfüllt wurde, dann geht es zur zweiten Vorbedingung. Deshalb wird diese zweite Vorbedingung mit ‚or else' bezeichnet.

3.3.3.3 Komplexe Vorbedingung der Substraktion (Komplexe Vorbedingung)

Die Komplexe Vorbedingung ist ähnlich wie die der Addition, wenn die einfache und mittlere Vorbedingung nicht erfüllt wurden, müssen alle Elemente der beiden Vektoren durch die Aux_Is.Substrakable kontrolliert werden, um Fehler und Zerstörung der Prozedur zu vermeiden.

```
procedure Subtract (Result : out Basic_Vector; Left, Right : in Basic_Vector)
   with
   Pre => (conds.AreValueSubstraktable1(Left  => Left, Right => Right)
            and then
            conds.AreValueSubstraktable2(Left  => Left, Right => Right))
                        or else
                                conds.PreSubtract(Left  => Left, Right => Right)
                                        or else
                                                (for all Index in Vector_Index =>
      (Aux.Is_Subtractable(Left.Vector_Values(Index), Right.Vector_Values(Index)))),
```

Die obigen Codes sind alle Vorbedingungen der Subtraktion der Vektoren.

3.3.3.4 Nachbedingung der Substraktion

Die Nachbedingung der Subtraktion ist ähnlich wie die der Addition. In der Nachbedingung der Addition muss man auch die mathematische Beziehung berücksichtigen. Das erzielte Ergebnis muss gleich wie das Ergebnis der Substraktion der beiden Vektoren sein und alle Ergebnisse müssen auch innerhalb der Grenze sein. Dann kann man programmieren:

```
post => (for all Index in Vector_Index =>
        (Result.Vector_Values(Index) = Left.Vector_Values(Index)-Right.Vector_Values(Index)))
                and then
                        conds.ResultinrangeSubstankt(Left  => Left,Right => Right);
```

ResultinrangeSubstankt ist eine Hilfsfunktion, die in Vector.conds programmiert wird. Mit Hilfe dieser Funktionen werden zwei eingegebene Vektoren subtrahiert und die erzielten Ergebnisse werden dann geprüft, ob sie alle in Ordnung sind.

3.3.4 Skalarprodukt

Skalarprodukt ist der schwerste Teil in dieser Arbeit, weil die Operation aus zwei mathematischen Vorgängen besteht. Am Anfang müssen alle Elemente der beiden Vektoren multipliziert werden, und danach müssen alle Ergebnisse nach der Multiplikation addiert werden. Es ist sehr schwer, die Vorbedingung für das Scala Produkt zu entwerfen, weil man gleichzeitig an die Addition und Multiplikation denken muss.
Jetzt sind die Grundlagen der mathematischen Norm bekannt und die Vorbedingungen im mathematischen Nachweis wurden erklärt. Für die Operation Scala Produkt wurden drei Vorbedingungen und zwei Nachbedingung entworfen.

3.3.4.1 ScalaProductPre1 (Einfache Vorbedingung von Skalarprodukt)

Im Skalarprodukt wurden insgesamt drei Vorbedingungen entworfen. ScalaProduct1 ist eine einfache Bedingung und wurde als Skalarprodukt programmiert.

Die einfache Vorbedingung des Skalarprodukts lautet:

```
function ScalaProductPre1(Left : in Basic_Vector; Right : in Basic_Vector) return Boolean is
Result : Boolean; Factor : Real_Type; ResultV : Real_Type;
begin
    Factor :=0.0; Result := True;
        for Index in Vector_Index loop
            if (Left.Vector_Values(Index) < Real_Type'Last) then
                Factor := Factor +1.0;
                        else
                                Factor := Factor;
            end if;
          end loop;
    ResultV:=  math.Sqrt( Factor * Factor * Left. Max_Norm * Left.Max_Norm * Right. Max_Norm *
                            Right.Max_Norm);
    Result := Result and (ResultV < Real_Type'Last);
    return Result;
  end ScalaProductPre1;
```

Dann wurde der obige Code erklärt. Nachdem der Vorgang Scala Produkt analysiert wurde, wurde folgendes ersichtlich, wenn die Maximalnorm der beiden Vektoren die Gleichung

$$\sqrt{n^2\left(\|a\|_\infty \|b\|_\infty\right)^2} \le Max\_\mathrm{Re}al$$

erfüllt hat, bleiben alle Elemente in den Vektoren und alle Ergebnis nach der Addition auch innerhalb der Grenze. Weil es nicht sehr streng ist, wurde es als einfache Vorbedingung verwendet. Am Anfang wurde eine Schleife programmiert, um den höchsten Index der Vektoren herauszufinden. Dann vergleicht man $\sqrt{n^2\left(\|a\|_\infty \|b\|_\infty\right)^2} \le Max\_\mathrm{Re}al$ mit der Grenze und am Ende wird ein Boolesches Ergebnis ausgegeben. Aber wenn die einfache Vorbedingung nicht erfüllt wurde, weiß man auch nicht, ob das Ergebnis in Ordnung ist oder überläuft, dann braucht man auch eine mittlere Vorbedingung.

3.3.4.2 ScalaProductPre2 (Mittlere Vorbedingung von Skalarprodukt)

Wenn die einfache Vorbedingung nicht erfüllt ist, braucht man auch eine mittlere Vorbedingung, um die eingegebenen Werte zu prüfen.

Die mittlere Vorbedingung wurde als Hilfsfunktion ScalaProductPre2 programmiert. Diese mittlere Vorbedingung prüft, ob ein Polynom aus dem Maximalwert der beiden Vektoren kleiner als die Grenze ist.

Dieses Polynom ist:

$$\frac{1}{4}\left[\left(a_{\max}+b_{\max}\right)^2 \times N - 0\right] \le Max\_\mathrm{Re}al$$

n bedeutet die höchste Reihe des Vektors. Man wählt dieses Polynom als die mittlere Vorbedingung, weil es strenger als das Polynom in der einfachen Vorbedingung ist. Wenn die Gleichung richtig erfüllt wurde, bedeutet dies, dass alle Ergebnisse nach der Berechnung in Ordnung sind.

Nachfolgend ist der Code von diesem Teil aufgeführt.

```
function ScalaProductPre2(Left : in Basic_Vector;Right : in Basic_Vector)return Boolean is
Result : Boolean; Factor : Real_Type;
ResultV : Real_Type; LeftMax,RightMax : Real_Type;
begin
    Factor :=0.0;Result := True;
        for Index in Vector_Index loop
            if (Left.Vector_Values(Index) < Real_Type'Last) then
                Factor := Factor +1.0;
                    else
                        Factor := Factor;
            end if;
        end loop;
    LeftMax := Vectors.Max(Left => Left);
    RightMax := Vectors.Max(Left => Right);
    ResultV :=  (0.25*(((LeftMax+ RightMax)*(LeftMax+RightMax))* Factor));

    Result := Result and (ResultV < Real_Type'Last);
    return Result;
end ScalaProductPre2;
```

Durch den obigen Code konnte folgendes festgestellt werden: Am Anfang wurde eine Schleife wie eine einfache Bedingung geschrieben, um die höchste Reihe des Vektors zu finden. Dann wurde dieses Polynom berechnet und das Ergebnis im ResultV zugewiesen. Am Ende wurden das erzielte Ergebnis und die Grenze miteinander verglichen, ob das erzielte Ergebnis überläuft. Am Ende wurde ein boolesches Ergebnis ausgegeben.

3.3.4.3 ScalaProductAll (Komplexe Vorbedingung von Skalarprodukt)

ScalaProductAll ist die komplexe Vorbedingung des Skalarprodukts. Es funktioniert, wenn die einfache und mittlere Vorbedingung nicht erfüllt sind. Die komplexe Vorbedingung ist auch die strengste Vorbedingung, deshalb müssen in diesem Schritt alle Elemente in beiden Vektoren geprüft werden, ob die Ergebnisse der Multiplikation und die Summe aller Ergebnisse der Multiplikation gültig sind.

Weil in Skalarprodukt Vorgänge aus Addition und Multiplikation bestehen, muss die Hilfsfunktion Is_Addable und Is_Scalable kombiniert werden. In einer Zeile kann man nicht diese zwei Funktionen zusammen verbinden, deshalb braucht man eine Hilfsfunktion, um dieses Ziel zu erreichen.

Der Code der komplexen Vorbedingung ist:

```
function ScalaProductPreAll(Left : in Basic_Vector; Right : in Basic_Vector) return Boolean is
 RightV : Real_Type; LeftV: Real_Type; ResultV : Real_Type;
 Result : Boolean; Summe : Real_Type;
          begin
              Result := True; Summe := 0.0;
                   for Index in Vector_Index loop
                            LeftV := Left.Vector_Values(Index);
                            RightV := Right.Vector_Values(Index);
                            ResultV := Left.Vector_Values(Index)* Right.Vector_Values(Index);
                            Result := Result and Aux.Is_Scalable(LeftV, RightV)
                                              and Aux.Is_Addable(Summe, ResultV);
                             Summe := Summe +ResultV;
          end loop;
return Result;
end ScalaProductPreAll;
```

Dann wird der obige Code erklärt, LeftV und RightV bezeichnet den Wert der beiden Vektoren. Im Multiplikationsteil muss geprüft werden, ob alle Ergebnisse der Multiplikation zwischen LeftV und RightV innerhalb der Grenze sind. Hier verwendet man die Hilfsfunktion

Is_Scalable, und im Befehl Aux.Is_Scalable(LeftV, RightV) werden alle Ergebnisse nach der Multiplikation geprüft. Um den Additionsteil zu prüfen, definiert man eine ResultV und Summe als Real_Type. ResultV bezeichnet das Ergebnis der Multiplikation, Summe bezeichnet die Summe der Ergebnisse der Multiplikation. Am Anfang setzt man im Ergebnis eine 0.0 ein. Mit Hilfe der Schleife treibt die Reihe der LeftV und RightV automatisch vor und wird ResultV immer erneuert. Das alte Ergebnis gibt man zur Summe hinzu und mit der Gleichung Summe:= Summe +ResultV werden sie zusammen addiert. Dann wird die Hilfsfunktion Aux.Is_Addable aufgerufen. Durch den Befehl Aux.Is_Addable(Summe, ResultV) wird die Summe der Multiplikation geprüft, ob sie alle gültig sind. Am Ende nutzt man eine ‚and' Beziehung, um zwei Befehle zu kombinieren und dann wird ein boolesches Ergebnis ausgegeben.

Nun wurden alle Vorbedingungen des Skalarprodukts ausgeführt. Jetzt müssen alle Vorbedingungen miteinander verbunden werden. Hier verwendet man die Beziehung ‚and then'. Nachfolgend ist der vollständige Code der Vorbedingungen des Skalarprodukts aufgeführt.

```
function Scalar_Product (Left, Right : in Basic_Vector; Error_Status : out Operation_Error_Type)
                        return Real_Type
  with
    pre => conds.ScalaProductPre1(Left  => Left, Right => Right)
              and then
              conds.ScalaProductPre2(Left  => Left, Right => Right)
              and then
              conds.ScalaProductPreAll(Left  => Left, Right => Right);
```

3.3.4.4 Nachbedingung des Skalarprodukts

Bei der Nachbedingung in Skalarprodukt muss auch die mathematische Beziehung der Vektoroperation berücksichtigt werden Aber wegen der komplizierten mathematischen Vorgänge kann man die Beziehung nicht direkt in einer Gleichung in die Nachbedingung programmieren. Hier musste eine Hilfsfunktion für die Nachbedingung entworfen werden und diese wurde als ScalaProductPost bezeichnet.

Hier folgt der Code von diesem Teil.

```
function ScalaProductPost(Left : in Basic_Vector; Right : in Basic_Vector) return Boolean is
RightV : Real_Type; LeftV: Real_Type; ResultV1, ResultV2 : Real_Type;
Result : Boolean; Summe : Real_Type; Err_Stat1: Vectors.Operation_Error_Type;
  begin
      Result := True; Summe := 0.0;
           for Index in Vector_Index loop
               LeftV := Left.Vector_Values(Index);
               RightV := Right.Vector_Values(Index);
                ResultV1 := Vectors.Scalar_Product(Left   => Left,
                                                   Right => Right,
                                                   Error_Status => Err_Stat1);
               ResultV2:= RightV* LeftV;
               Summe := Summe + ResultV2;
               Result := Result and aux.Is_Zero( Summe- ResultV1);

           end loop;
    return Result;
  end ScalaProductPost;
```

In dieser Hilfsfunktion wurde ein mathematischer Vorgang wie Skalarprodukt entworfen. LeftV und RightV sind die Werte der beiden Vektoren. LeftV und RightV wurden zuerst multipliziert und dann die Ergebnisse nach der Multiplikation addiert. Am Ende erhielt man ein Ergebnis als Summe, die als Real_Type definiert wurde. Dann wurde die Funktion Skalarprodukt aufgerufen und diese gleichen Vektoren eingegeben. Dann wurde das Ergebnis als ResultV1 im Real_Type ausgegeben. Am Ende verwendete man die Hilfsfunktion Aux.Is_Zero. Diese Funktion kann testen, ob das Ergebnis zwischen zwei Werten 0.0 ist. Hier wurde die Summe und ResultV1 eingegeben. Wenn das Skalarprodukt richtig ausgeführt wird, müssen der Wert der Summe und ResultV1 gleich sein. Dies führt zu dem Ergebnis der Subtraktion zwischen Summe und ResultV1 und muss auch 0.0 sein. Dann kann man feststellen, ob das Skalarprodukt richtig ausgeführt wurde.

Dann können die Nachbedingungen des Skalarprodukts programmiert werden.

```
post =>conds.ScalaProductPost(Left  => Left, Right => Right)
          and then
         conds.ResultinrangeScalaProduct(Left  => Left, Right => Right) ;
```

ResultinrangeScalaProduct ist auch eine Hilfsfunktion, die überprüft, ob das erzielte Ergebnis in Ordnung ist.

3.4 Programmierung des Testteils der Vorbedingung

In Spark gibt es keine Testprogrammfunktionen. Das bedeutet, wenn das Programm erstellt wurde, muss der Benutzer es in der Praxis testen. Das ist wirklich schlimm. Aber in Ada kann das Testprogramm für die Vor- und Nachbedingungen entworfen werden. In Ada muss geprüft werden, was passiert, wenn die Vorbedingung nicht richtig ausgeführt und wenn unterschiedliche Werte der zwei Vektoren eingegeben wurden. Ist das erzielte Ergebnis richtig?
Beim Scheitern des Tests der Vorbedingung kann man folgendes programmieren:

```
declare
        pragma Assertion_Policy(Check);
        FoundError : Boolean := False;
     begin
           ops.Add(ResultV,LeftV,RightV);
      exception
            when  Constraint_Error => FoundError:= True;

      AUnit.Assertions.Assert
         (FoundError,
             "Overflow bei Zuweisung");
end;
```

pragma Assertion_Policy(Check) ist ein wichtiger Befehl im Testteil. Normalerweise ist die Vorbedingung und Nachbedingung zum Testteil nicht sichtbar. Mit folgendem Befehl kann der Vorbedingungsteil sichtbar gemacht werden und dann kann man testen. Constraint_Error ist ein systemdefinierter Fehlertyp, er hat die gleiche Bedeutung wie Overflow_Error.

Am Anfang wurde eine Boolean Typ Variable ‚FoundError' als ‚False' Zustand definiert, dann wurde die getestete Vektoroperation ausgeführt. Beim obigen Code wurde die Hilfsfunktion der Add von Vector.ops aufgerufen. Nach dem Befehl Begin mussten die Werte zu beiden Vektoren eingeben werden. Anschließend konnte man kontrollieren, ob das erzielte Ergebnis einen richtigen Zustand besitzt.

Bei der Eingabe der Werte zu den Vektoren gab es ein Prinzip, man musste diese Situationen bei den Vorbedingungen testen.

- Was passiert, wenn die einfache Vorbedingung erfüllt wurde?
- Was passiert, wenn die einfache Vorbedingung scheitert, aber die mittlere Vorbedingung erfüllt wurde?
- Was passiert, wenn die einfache und mittlere Vorbedingung scheitert, aber die komplexe Vorbedingung richtig erfüllt wurde?.
- Was passiert, wenn alle drei Stufen der Vorbedingung nicht erfüllt wurden?

Wenn das Ergebnis Überlauffehler auftrat, verändert ‚FoundError' von ‚False' bis ‚True'. Dann wurde der Zustand der ‚FoundError' am Ende kontrolliert, ob ein richtiges boolesches Ergebnis ausgegeben wurde.
Hier gibt es drei Vorgänge als Beispiele für den Test-Anteil, es sind Addition, Multiplikation und Skalarprodukt. Der Testvorgang der Addition ist ähnlich wie der der Subtraktion, dann kann man zusammenfassen. Der Vorgang des Skalarprodukts ist komplizierter, als der der anderen drei Vektoroperationen, deshalb wird nachfolgend der Vorgang der Addition, Multiplikation und Skalarprodukt erläutert.

3.4.1 Testvorgang der Addition

3.4.1.1 Test der einfachen Vorbedingung der Addition

Die Vorbedingung der Addition besteht aus drei Teilen, es sind die einfache, mittlere und komplexe Vorbedingung. Wenn man die Vorbedingung der Addition testen möchte, muss man erst die einfache Vorbedingung testen. Was passiert, wenn die Vorbedingung der Addition erfüllt wurde.

```
declare
   pragma Assertion_Policy(Check);
   FoundError : Boolean := False;

begin
        RightX.Set_Element_At(Element_Value => 0.1*Real_Type'Last,   At_Index     => 1);
        RightX.Set_Element_At(Element_Value => 0.2*Real_Type'Last,   At_Index     => 2);
        RightX.Set_Element_At(Element_Value => 0.4*Real_Type'Last,   At_Index     => 3);
        LeftX.Set_Element_At(Element_Value => 0.6*Real_Type'Last,    At_Index     => 1);
        LeftX.Set_Element_At(Element_Value => 0.5*Real_Type'Last,    At_Index     => 2);
        LeftX.Set_Element_At(Element_Value => 0.4*Real_Type'Last,    At_Index     => 3);

        ops.add(Result => ResultX,  Left   => LeftX, Right  => RightX);
exception
   when  Constraint_Error => FoundError:= True;

 AUnit.Assertions.Assert
      (Not FoundError,
           "keine Overflow bei Zuweisung");
end;
```

Der obige Code war die erste Prüfung. Man musste jedes der drei Elemente zwei Vektoren zuweisen. Bei diesen Werten wurde bereits im mathematischen Beweis erklärt, dass damit nur die einfache Vorbedingung richtig erfüllt werden kann.
Dann wurde die Funktion Addition von Vektor.ops aufgerufen. Es wurde definiert, wenn ein Überlauffehler auftritt, verändert sich der Zustand der Vektoren von ‚False' zu ‚True'. Weil die Summe in diesem Schritt in Ordnung ist, bekommt die boolesche Variable ‚FoundError' einen ‚False' Zustand. Deshalb wurde am Ende getestet, ob das ‚FoundError' wirklich einen richtigen Zustand bekommt.

3.4.1.2 Test der mittleren Vorbedingung der Addition

Dann ging es zur zweiten Prüfung. Was wird passieren, wenn die einfache Vorbedingung nicht erfüllt, aber die mittlere Vorbedingung erfüllt wurde? Der ganze Testvorgang ist ähnlich wie der der ersten Prüfung und es musste nur der eingegebene Wert verändert werden.

```
-- 2. Einfache Vorbedingung der Addition nicht erfüllt, aber Mittlere Vorbedingung der Addition erfüllt
   ResultX.Initialize;
   LeftX.Initialize;
   RightX.Initialize;

   declare
     pragma Assertion_Policy(Check);
     FoundError : Boolean := False;

   begin
         RightX.Set_Element_At(Element_Value => 0.1*Real_Type'Last, At_Index     => 1);
         RightX.Set_Element_At(Element_Value => 0.2*Real_Type'Last, At_Index     => 2);
         RightX.Set_Element_At(Element_Value => 0.5*Real_Type'Last, At_Index     => 3);
         LeftX.Set_Element_At(Element_Value => 0.6*Real_Type'Last,   At_Index     => 1);
         LeftX.Set_Element_At(Element_Value => 0.5*Real_Type'Last,   At_Index     => 2);
         LeftX.Set_Element_At(Element_Value => 0.4*Real_Type'Last,   At_Index     => 3);

         ops.add(Result => ResultX, Left   => LeftX, Right  => RightX);
    exception
      when  Constraint_Error => FoundError:= True;

    AUnit.Assertions.Assert
         (Not FoundError,
            "keine Overflow bei Zuweisung");

   end;
```

Der obige Code gehört zur zweiten Prüfung. Es wurden nur die eingegeben Werte der zwei Vektoren verändert. Die einfache Vorbedingung ist, wenn die Summe der Maximalnorm der beiden Vektoren kleiner als die Grenze ist, kann die Addition richtig ausgeführt werden. Die Maximalnorm, der hier eingegebenen Vektoren, ist 0.5*Real_Type'Last und 0.6*Real_Type'Last. Die Summe beträgt 1.1*Real_Type'Last, es läuft schon die Grenze über. Aber wenn man berechnet, das Ergebnis ist (0.7*Real_Type'Last, 0.7*Real_Type'Last,

0.9*Real_Type'Last). Alle drei Elemente sind in Ordnung. Obwohl es die einfache Vorbedingung nicht bestand, wird es durch die mittlere Vorbedingung weiter geprüft.

3.4.1.3 Test der komplexen Vorbedingung der Addition

Dann ging es zur dritten Prüfung, dass die einfache und mittlere Vorbedingung beide nicht erfüllt wurden, aber die komplexe Vorbedingung richtig erfüllt war. Hier mussten noch die geeigneten Werte für zwei Vektoren ausgewählt werden.

```
– 3. Einfache und Mittlere Vorbedingungen der Addition nicht erfüllt, aber komplexe Vorbedingung der Addition erfüllt

   declare
      pragma Assertion_Policy(Check);
      FoundError : Boolean := False;

   begin
            RightX.Set_Element_At(Element_Value => 0.1*Real_Type'Last, At_Index     => 1);
            RightX.Set_Element_At(Element_Value => 0.2*Real_Type'Last, At_Index     => 2);
            RightX.Set_Element_At(Element_Value => 0.7*Real_Type'Last, At_Index     => 3);
            LeftX.Set_Element_At(Element_Value => 0.6*Real_Type'Last, At_Index     => 1);
          LeftX.Set_Element_At(Element_Value => 0.5*Real_Type'Last, At_Index     => 2);
          LeftX.Set_Element_At(Element_Value => 0.5*Real_Type'Last, At_Index     => 3);

           ops.add(Result => ResultX, Left   => LeftX, Right  => RightX);
     exception
           when  Constraint_Error => FoundError:= True;
     AUnit.Assertions.Assert
          (FoundError,
               "Overflow bei Zuweisung ");
```

Jetzt wurden die eingegebenen Werte von obigem Code analysiert. Die Summe der Maximalenorm der beiden Vektoren beträgt 1.3*Real_Type'Last, deshalb wurde die einfache Vorbedingung nicht erfüllt. Dann wurde die Reihe, deren Wert größer als 0.5*Real_Type'Last ist, automatisch herausgesucht. In diesen Vektoren mussten alle drei Reihen kontrolliert werden. Und das Ergebnis läuft auch über, deshalb erfüllt es auch nicht die mittlere Vorbedingung. Dann wurde es durch die komplexe Vorbedingung geprüft.

Danach kam die vierte Prüfung. Was wird passieren, wenn alle Vorbedingungen nicht erfüllt wurden? Es tritt am Ende ein Überlauffehler auf.

```
--4. Einfache, Mittlere und Komplexe Vorbedingungen sind alle nicht erfüllt
   declare
     pragma Assertion_Policy(Check);
     FoundError : Boolean := False;
   begin
         RightX.Set_Element_At(Element_Value => 0.4*Real_Type'Last,  At_Index     => 1);
         RightX.Set_Element_At(Element_Value => 0.5*Real_Type'Last,  At_Index     => 2);
         RightX.Set_Element_At(Element_Value => 0.9*Real_Type'Last,  At_Index     => 3);
         LeftX.Set_Element_At(Element_Value => 0.9*Real_Type'Last,   At_Index     => 1);
         LeftX.Set_Element_At(Element_Value => 0.5*Real_Type'Last,   At_Index     => 2);
         LeftX.Set_Element_At(Element_Value => 0.8*Real_Type'Last,   At_Index     => 3);

         ops.add(Result => ResultX, Left   => LeftX, Right  => RightX);
  exception
         when  Constraint_error => FoundError:= True;

    AUnit.Assertions.Assert
     (FoundError,
           "Overflow bei Zuweisung ");
   end;
```

Die Summe der zwei Vektoren beträgt 1.8*Real_Type'Last, deshalb erfüllt sie nicht die einfache Vorbedingung. Je nach der mittleren Vorbedingung mussten alle Reihen der Vektoren kontrolliert werden, aber das Ergebnis läuft auch über. Zum Schluss wurde die komplexe Vorbedingung der Addition ausgeführt, die Addition aller Werte in beiden Vektoren. Aber das Ergebnis läuft wieder über. Dann kann diese Addition nicht ausgeführt werden und ein Überlauffehler trat am Ende auf.

3.4.2 Testvorgang der Multiplikation

3.4.2.1 Test der einfachen Vorbedingung der Multiplikation

Beim Test der Vorbedingung der Multiplikation ist es relativ einfach, weil man nur an einen Vektor denken musste.

Am Anfang wurde die einfache Vorbedingung getestet.

```
declare
      pragma Assertion_Policy(Check);
      FoundError : Boolean := False;

   begin
        factor :=2.0 ;
        LeftX.Set_Element_At(Element_Value => 0.1*Real_Type'Last, At_Index     => 1);
        LeftX.Set_Element_At(Element_Value => 0.4*Real_Type'Last, At_Index     => 2);
        LeftX.Set_Element_At(Element_Value => 0.2*Real_Type'Last, At_Index     => 3);

        ops.Scale(Result => ResultX, Left   => LeftX, Factor => factor);

   exception
      when   Constraint_Error => FoundError:= True;

    AUnit.Assertions.Assert
     (Not FoundError,
          "keine Overflow bei Zuweisung");
   end;
```

Der obige Code zeigt den Test der einfachen Vorbedingung der Multiplikation. Am Anfang wurde ein Boolesches Ergebnis definiert. Dann wurde der geeignete Wert zur Koeffizient und zum Vektor gegeben. Die hier entworfene einfache Vorbedingung der Multiplikation lautet, wenn das Ergebnis der Multiplikation der Maximalnorm und Koeffizient kleiner als die Grenze ist, kann die Multiplikation richtig ausgeführt werden. Die Maximalnorm des Vektors ist 0.4*Real_Type'Last. Das Ergebnis beträgt 0.8*Real_Type'Last und es ist in Ordnung. Deshalb wurde die einfache Vorbedingung erfüllt und das boolesche Ergebnis bekam am Ende einen ‚False' Zustand.

3.4.2.2 Test der mittleren Vorbedingung der Multiplikation

Dann musste man an die mittlere Vorbedingung denken. Weil der Multiplikationsvorgang relativ einfach ist, konnten fast alle Vektoren der einfachen Vorbedingung geprüft werden.

Dann braucht man eine mittlere Vorbedingung, die alle Elemente im Vektor testen kann.

```
declare
      pragma Assertion_Policy(Check);
      FoundError : Boolean := False;
begin
    factor :=2.0 ;
    LeftX.Set_Element_At(Element_Value => 0.1*Real_Type'Last, At_Index     => 1);
    LeftX.Set_Element_At(Element_Value => 0.6*Real_Type'Last, At_Index     => 2);
    LeftX.Set_Element_At(Element_Value => 0.2*Real_Type'Last, At_Index     => 3);

      ops.Scale(Result => ResultX, Left   => LeftX, Factor => factor);

exception
      when   Constraint_Error => FoundError:= True;

AUnit.Assertions.Assert
     (FoundError, "Overflow bei Zuweisung ");

end;
```

Die Maximalnorm des Vektors im obigen Code ist 0.6*Real_Type'Last. Das Ergebnis zwischen der Maximalnorm und Koeffizient beträgt 1.2*Real_Type'Last. Es läuft schon über und dann wurden alle Elemente im Vektor noch einmal geprüft. Am Ende erhielt das boolesche Ergebnis einen ‚True' Zustand.

3.4.3 Test des Skalarprodukts

3.4.3.1 Test der einfachen Vorbedingung des Skalarprodukts

Der Vorgang des Skalarprodukts ist relativ kompliziert, weil er aus Additions- und Multiplikationsvorgang besteht. Für diesen Teil musste zuerst die mathematische Grundlage einer einfachen Vorbedingung entworfen werden. Die Maximalnorm der beiden Vektoren muss die Gleichung

Gleichung 1: 3.4.3.1

$$\sqrt{n^2\left(\|a\|_\infty \|b\|_\infty\right)^2} \leq Max\_\mathrm{Re}al$$

erfüllen.

Es braucht die Maximalnorm der beiden Vektoren und die höchste Reihe des Vektors.

```
declare
      pragma Assertion_Policy(Check);
      FoundError : Boolean := False;
      Error_Stat : Vectors.Operation_Error_Type;

      begin
         LeftX.Set_Element_At(Element_Value => 0.1*math.Sqrt(Real_Type'Last), At_Index     => 1);
         LeftX.Set_Element_At(Element_Value => 0.2*math.Sqrt(Real_Type'Last), At_Index     => 2);
         LeftX.Set_Element_At(Element_Value => 0.3*math.Sqrt(Real_Type'Last), At_Index     => 3);
         RightX.Set_Element_At(Element_Value => 0.4*math.Sqrt(Real_Type'Last), At_Index     => 1);
         RightX.Set_Element_At(Element_Value => 0.1*math.Sqrt(Real_Type'Last),At_Index     => 2);
         RightX.Set_Element_At(Element_Value => 0.1*math.Sqrt(Real_Type'Last),At_Index     => 3);

         ResultX := Vectors.ops.Scalar_Product(Left=> LeftX, Right => RightX,
                                               Error_Status => Error_Stat);
   exception
     when   Constraint_Error => FoundError:= True;

    AUnit.Assertions.Assert
     (Not FoundError,
          "Overflow bei Zuweisung ");
```

Die Maximalnorm der zwei Vektoren ist 0.3*Wurzel(Real_Type'Last) und 0.4*Wurzel(Real_Type'Last). n bedeutet, wie viele Elemente jeder Vektor besitzt. Hier wurden jeweils drei Elemente den Vektoren zugewiesen, deshalb beträgt n hier 3. Das Ergebnis

nach der Berechnung der Gleichung ist 0.09*Real_Type'Last(In Ordnung). Deshalb hat es die einfache Vorbedingung erfüllt und bekommt ‚FoundError' am Ende ein richtiger Zustand.

3.4.3.2 Test der mittleren Vorbedingung des Skalarprodukts

Dann musste die mittlere Vorbedingung getestet werden. Hier wurden zwei Vorbedingungen definiert und jede mittlere Vorbedingung brauchte einen Test. Für die erste mittlere Vorbedingung galt, wenn ein Wert des Vektors größer als die Wurzel der Grenze ist, ergab sich die Möglichkeit, die Multiplikation dieser Reihe zu prüfen.

Die zweite Vorbedingung war, dass die Werte im Vektor die Gleichung

Gleichung 2: 3.4.3.2

$$\frac{1}{4}\left[\left(a_{\max}+b_{\max}\right)^2 \times N - 0\right] \leq Max\_Real$$

erfüllen sollen. Diese zwei mittleren Vorbedingungen müssen gleichzeitig erfüllt sein.
Weil die Grundstruktur ähnlich ist, wird hier nur die Veränderung der Werte gezeigt und erläutert.

```
begin
    LeftX.Set_Element_At(Element_Value => 0.5*math.Sqrt(Real_Type'Last),At_Index     => 1);
    LeftX.Set_Element_At(Element_Value => 0.3*math.Sqrt(Real_Type'Last),At_Index     => 2);
    LeftX.Set_Element_At(Element_Value => 0.4*math.Sqrt(Real_Type'Last),At_Index     => 3);
    RightX.Set_Element_At(Element_Value => 0.1*math.Sqrt(Real_Type'Last),At_Index     => 1);
    RightX.Set_Element_At(Element_Value => -0.7*math.Sqrt(Real_Type'Last),At_Index     => 2);
    RightX.Set_Element_At(Element_Value => 0.5*math.Sqrt(Real_Type'Last),At_Index     => 3);
    ResultX := Vectors.ops.Scalar_Product(Left         => LeftX,
                                          Right        => RightX,
                                          Error_Status => Error_Stat);
```

Durch den obigen Code kann man ersehen, dass alle Werte der Vektoren kleiner als die Wurzel der Grenze sind, deshalb erfüllt er die einfache Vorbedingung. Nach der Berechnung der Gleichung, beträgt das Ergebnis 0.04*Real_Type'Last. Es läuft nicht über, deshalb erfüllt es beide mittlere Vorbedingungen und das ‚FoundError' bekommt am Ende einen ‚False' Zustand.

Dann war es noch notwendig, die erste mittlere Vorbedingung zu testen. Danach wurden unterschiedliche Werte zu beiden Vektoren eingegeben.

```
Begin
    LeftX.Set_Element_At(Element_Value => 1.0*math.Sqrt(Real_Type'Last), At_Index     => 1);
    LeftX.Set_Element_At(Element_Value => 0.1*math.Sqrt(Real_Type'Last),At_Index     => 2);
    LeftX.Set_Element_At(Element_Value => 1.0*math.Sqrt(Real_Type'Last), At_Index     => 3);
    RightX.Set_Element_At(Element_Value => 0.1*math.Sqrt(Real_Type'Last),At_Index     => 1);
    RightX.Set_Element_At(Element_Value => -2.0*math.Sqrt(Real_Type'Last),At_Index     => 2);
    RightX.Set_Element_At(Element_Value => 0.1*math.Sqrt(Real_Type'Last),At_Index     => 3);
    ResultX := Vectors.ops.Scalar_Product(Left        => LeftX,
                                          Right       => RightX,
                                          Error_Status => Error_Stat);
```

Diese zwei Vektoren wurden erst von der ersten mittleren Vorbedingung kontrolliert. Die zweite Reihe hat den Wert -2.0*Wurzel(Real_Type'Last), größer als Wurzel(Real_Type'Last), deshalb wird das Element in dieser Reihe multipliziert. Das Ergebnis beträgt -0.2*Real_Type'Last, es bleibt in der Grenze. Das Ergebnis nach der Berechnung der Gleichung ist 0.9575*Real_Type'Last. Diese zwei Vektoren haben die beiden mittleren Vorbedingungen erfüllt. Am Ende bekommt das ‚FoundError' auch einen ‚False' Zustand.

3.4.3.3 Test der komplexen Vorbedingung des Skalarprodukts

Nach dem Test aller mittleren Vorbedingungen musste noch an die komplexe Vorbedingung gedacht werden. Die komplexe Vorbedingung ist die gleiche wie die der anderen Vektoroperation, dass alle Elemente in den Vektoren berechnet werden. Hier musste noch über zwei Möglichkeiten nachgedacht werden.

- Was passiert, wenn die einfache Vorbedingung erfüllt, aber die mittlere Vorbedingung nicht erfüllt ist?
- Was passiert, wenn die einfache und die mittlere Vorbedingung beide nicht erfüllt sind?

Jetzt wird mit der ersten Möglichkeit begonnen.

```
Begin
    LeftX.Set_Element_At(Element_Value => 0.1*math.Sqrt(Real_Type'Last),At_Index     => 1);
    LeftX.Set_Element_At(Element_Value => 0.2*math.Sqrt(Real_Type'Last),At_Index     => 2);
    LeftX.Set_Element_At(Element_Value => 1.0*math.Sqrt(Real_Type'Last),At_Index     => 3);
    RightX.Set_Element_At(Element_Value => 1.0*math.Sqrt(Real_Type'Last),At_Index     => 1);
```

```
RightX.Set_Element_At(Element_Value => 2.0*math.Sqrt(Real_Type'Last),At_Index     => 2);
RightX.Set_Element_At(Element_Value => 0.5*math.Sqrt(Real_Type'Last),At_Index     => 3);

ResultX := Vectors.ops.Scalar_Product(Left        => LeftX,
                                      Right       => RightX,
                                      Error_Status => Error_Stat);
```

Der obige Code ist die erste Möglichkeit. Nach der ersten Vorbedingung mussten alle Elemente in den Vektoren geprüft werden. Das Ergebnis der ersten Vorbedingung ist in Ordnung. Aber nach der Berechnung (zweite Vorbedingung) beträgt eine 6.75*Real_Type'Last. Es hat die erste mittlere Vorbedingung bestanden, aber die zweite Vorbedingung nicht erfüllt. Deshalb wurden alle Elemente durch die komplexe Vorbedingung noch einmal geprüft.
Nun geht es zur zweiten Möglichkeit:

```
begin
    LeftX.Set_Element_At(Element_Value => 1.0*math.Sqrt(Real_Type'Last),At_Index     => 1);
    LeftX.Set_Element_At(Element_Value => 0.3*math.Sqrt(Real_Type'Last),At_Index     => 2);
    LeftX.Set_Element_At(Element_Value => 1.0*math.Sqrt(Real_Type'Last),At_Index     => 3);
    RightX.Set_Element_At(Element_Value => 0.5*math.Sqrt(Real_Type'Last),At_Index     => 1);
    RightX.Set_Element_At(Element_Value => 0.5*math.Sqrt(Real_Type'Last),At_Index     => 2);
    RightX.Set_Element_At(Element_Value => 0.3*math.Sqrt(Real_Type'Last),At_Index     => 3);

    ResultX := Vectors.ops.Scalar_Product(Left        => LeftX,
                                          Right       => RightX,
                                          Error_Status => Error_Stat);
```

Die zwei eingegebenen Vektoren wurden zuerst durch die einfache Vorbedingung kontrolliert. Wenn man die Vektoren in die Gleichung legt, bekommt man folgendes Ergebnis:

$$\sqrt{n^2\left(\|a\|_\infty \|b\|_\infty\right)^2} = \sqrt{3^2\left(\sqrt{R}\times 0.5\sqrt{R}\right)^2} = 1.5R$$

Es läuft schon über, deshalb erfüllte es die einfache Vorbedingung nicht.
Dann wurden diese Vektoren durch die mittlere Vorbedingung geprüft.

Das Ergebnis lautet:

$$\frac{1}{4}\left[\left(a_{\max}+b_{\max}\right)^2\times N-0\right]=\frac{3}{4}\left(\sqrt{R}+0.5\sqrt{R}\right)^2=1.6875R>R$$

Das Ergebnis nach der Berechnung der mittleren Vorbedingung läuft auch über, deshalb wurden die einfache und mittlere Vorbedingung beide nicht erfüllt. Diese zwei Vektoren wurden durch die komplexe Vorbedingung noch einmal exakter geprüft. Das Ergebnis nach der Berechnung der komplexen Vorbedingung beträgt:

0.5R + 0.15R + 0.3R=0.95R

Weil das Ergebnis in Ordnung ist, ist der ‚FoundError' am Ende ein ‚False' Zustand.

Am Ende musste wegen der Situation, dass alle drei Vorbedingungen nicht erfüllt waren, getestet werden. Hier wurden zwei neue Vektoren erzeugt.

```
begin
    LeftX.Set_Element_At(Element_Value => 0.1*math.Sqrt(Real_Type'Last), At_Index     => 1);
    LeftX.Set_Element_At(Element_Value => 0.2*math.Sqrt(Real_Type'Last), At_Index     => 2);
    LeftX.Set_Element_At(Element_Value => 2.0*math.Sqrt(Real_Type'Last),At_Index     => 3);
    RightX.Set_Element_At(Element_Value => 2.0*math.Sqrt(Real_Type'Last),At_Index     => 1);
    RightX.Set_Element_At(Element_Value => 0.5*math.Sqrt(Real_Type'Last),At_Index     => 2);
    RightX.Set_Element_At(Element_Value => 0.4*math.Sqrt(Real_Type'Last),At_Index     => 3);

    ResultX := Vectors.ops.Scalar_Product(Left=> LeftX,
                                          Right  => RightX,
                                          Error_Status => Error_Stat);
```

Am Anfang wurden die Vektoren durch die einfache Vorbedingung geprüft, das erzielte Ergebnis beträgt 4*Real_Type'Last. Deshalb erfüllte es die einfache Vorbedingung nicht. Dann wurden die Vektoren durch die mittlere Vorbedingung getestet:

$$\frac{1}{4}\left[\left(a_{\max}+b_{\max}\right)^2\times N-0\right]=\frac{3}{4}\left(2\sqrt{R}+2\sqrt{R}\right)^2=12R>R$$

Das Ergebnis nach der mittleren Vorbedingung beträgt 12*Real_Type'Last, deshalb erfüllte es die mittlere Vorbedingung auch nicht. Zum Schluss wurde sie durch die komplexe Vorbedingung getestet.

$$a_1 b_1 + a_2 b_2 + ... + a_N b_N = 0.2R + 0.1R + 0.8R = 1.1R > R$$

Das Ergebnis nach dem Skalarprodukt beträgt 1.1*Real_Type'Last. Ein Überlauffehler trat auf, deshalb bekommt der ‚FoundError' am Ende einen ‚True' Zustand.

4 Zusammenfassung und Ausblick

Im dieser Bachelorarbeit wurden mit Hilfe von Prof. Enzmann die Vor- und Nachbedingungen der Operationen für die arithmetische Berechnung zwischen Vektoren entworfen. Die erste Schwierigkeit bestand im Lernen von Entwurf und Kombination der Hilfsfunktion. Am Anfang wurde versucht, alle Inhalte der Vorbedingung und Nachbedingung in einer Spezifikationsdatei zu programmieren, aber es sah nicht gut aus und war auch sehr fehlerhaft. Um die Sauberkeit des Programmes halten zu können, wurden viele Hilfsfunktionen entworfen und dann alle Hilfsfunktionen kombiniert. Eine hohe Sauberkeit und Sicherheit des Programms konnte so garantiert werden.

Am Anfang des Entwurfs der Vor- und Nachbedingung trat die Schwierigkeit auf, dass der Fehler, Circular Unit Dependency immer passierte. Zu Beginn wurden alle Hilfsfunktionen im Vektor.aux geschrieben und man glaubte, im Vector.ads können alle Hilfsfunktionen aufgerufen und miteinander kombiniert werden. Aber weil Vector.aux ein Kindpaket des Vektors ist, sind diese beiden Dateien voneinander abhängig, deshalb trat immer diese Circular Unit Dependency auf. Mit der Hilfe von Prof. Enzmann wurde die Grundstruktur verändert und alle mathematischen Vektoroperationen in Vector.ops und alle Hilfsfunktionen in Vector.conds programmiert. Dann wurden in Vector.ops die Hilfsfunktionen in Vector.conds aufgerufen. Sie sind nicht voneinander abhängig, deshalb konnten alle Vor- und Nachbedingungen richtig programmiert und verbunden werden.

Nachdem alle Hilfsfunktionen geschrieben waren, wurde versucht, das Testprogramm zu schreiben. Um die Robustheit des Programms zu bestimmen, sollten alle entworfenen Vektoroperationen getestet werden. Am Anfang wurden die Vor- und Nachbedingungen, die entworfen wurden, ernsthaft analysiert. In diesem Teil sollte an verschiedenen Beispielen jede Vektoroperation getestet werden, um die Richtigkeit des entwickelten Programms prüfen zu können. Mit Hilfe der mathematischen Bücher konnten geeignete Beispiele für jede Vorbedingung entworfen werden. Es wurde jede einfache, mittlere und komplexe Vorbedingung jeder Vektoroperation analysiert und geeignete Beispiele für jede Stufe der Vorbedingung gesucht.

Die Bibliothek wurde meistens auf der Berechnung von Signal, z. B. Faltung verwendet. Mit der Hilfe der Vor- und Nachbedingung konnten eine höhere Sauberkeit und Sicherheit des Vorgangs der Bearbeitung von Vektoren realisiert werden. Es konnten auch viele Prüfschritte gespart werden und mit Hilfe der drei Etagen Vorbedingung konnte die Prozedur in der Laufzeit exakter die Werte in den eingegebenen Vektoren kontrollieren. Die Vorbedingung und Nachbedingung, die für jede Vektoroperationen entworfen wurde, gehörte am Ende zur Bibliothek: Vektoroperation. Angesichts der Komplexität der Hardware - Umgebung und der

nicht wirklich einheitlichen Standardbibliothek bestand auch das Risiko bei der Benutzung der Standardbibliothek. Die im Verlauf dieser Arbeit entworfene Bibliothek konnte das Risiko möglichst vermeiden. Die Flexibilität und Robustheit war sehr stabil, und durch die Leistung der Prüfschritte wurde auch die exakte Vor- und Nachbedingung gespart.

Anhang 1: Quellen- und Literaturverzeichnis

Buchquellen

[1] John Barnes: High Integrity Software-The SPARK Approach to Safety and Security (Seite 93)

[2] Ian Sommerville: Software Engineering (Seite 46, 397, 402)

[3] Steve Oualline,O'Reilly Germany, 2004: Praktische C++ - Programmierung (Seite 415)

Anhang 2: Abbildungsverzeichnis

Anhang 3: Tabellenverzeichnis

Anhang 4: Gleichungsverzeichnis