SPARK95

Ingmar Wirths

12. Juli 2007

Motivation

- Ada wurde zur Programmierung von Mikroprozessoren entwickelt.

Motivation

- Ada wurde zur Programmierung von Mikroprozessoren entwickelt.
- Ein Systemversagen ist oft nicht tolerierbar.

Motivation

- Ada wurde zur Programmierung von Mikroprozessoren entwickelt.
- Ein Systemversagen ist oft nicht tolerierbar.
- Industrie und Militär verlangen für kritische Systeme Korrektheitsbeweise.

Einleitung

- SPARK ist eine annotierte Teilsprache von Ada.

Einleitung

- SPARK ist eine annotierte Teilsprache von Ada.
- SPARK Annotations sind Ada Kommentare: $--\#\ foo;$

Einleitung

- SPARK ist eine annotierte Teilsprache von Ada.
- SPARK Annotations sind Ada Kommentare: $--\#\ foo;$
- In SPARK geschriebene Programme können von jedem Ada Compiler übersetzt werden.

Einleitung

- SPARK ist eine annotierte Teilsprache von Ada.
- SPARK Annotations sind Ada Kommentare: $--\#\ foo;$
- In SPARK geschriebene Programme können von jedem Ada Compiler übersetzt werden.
- Einige Sprachkonstrukte aus Ada sind verboten.

Correctness by Construction

- Ada unterscheidet zwischen Kompilier- und Laufzeitfehlern

Correctness by Construction

- Ada unterscheidet zwischen Kompilier- und Laufzeitfehlern
- Das Ziel von SPARK ist es Programme zu schreiben, die keine Laufzeitfehler produzieren.

Correctness by Construction

- Ada unterscheidet zwischen Kompilier- und Laufzeitfehlern
- Das Ziel von SPARK ist es Programme zu schreiben, die keine Laufzeitfehler produzieren.
- Dazu wird der Programmcode mit dem SPARK Toolset einer statischen Analyse unterzogen.

Correctness by Construction

- Ada unterscheidet zwischen Kompilier- und Laufzeitfehlern
- Das Ziel von SPARK ist es Programme zu schreiben, die keine Laufzeitfehler produzieren.
- Dazu wird der Programmcode mit dem SPARK Toolset einer statischen Analyse unterzogen.
- SPARK kann Daten- und Informationsflussanalyse sowie Programmverifikation durchführen.

Datenflussanalyse

- Variablen müssen vor Benutzung initialisiert werden.

Datenflussanalyse

- Variablen müssen vor Benutzung initialisiert werden.
- Modus von Parametern (**in**, **out**, **in out**)

Datenflussanalyse

- Variablen müssen vor Benutzung initialisiert werden.
- Modus von Parametern (**in**, **out**, **in out**)
- Annotation einer globalen Variable in einer Prozedur:
  $--\#$ *global foo;*

Informationsflussanalyse

- Abhängigkeiten von Variablen werden mit $--\#$ *derives X from X, Y;* deklariert.

Informationsflussanalyse

- Abhängigkeiten von Variablen werden mit
  `--# derives X from X, Y;` deklariert.
- Beispiel:

```
procedure Increment (X : in out Counter_Type);
--# global Counter;
--# derives X from X, Counter;
```

Programmverifikation

- Zusicherungen $--\#$ *assert*

Programmverifikation

- Zusicherungen $--\#$ *assert*
- Vor- und Nachbedingungen werden mit $--\#$ *pre* bzw. $--\#$ *post* deklariert.

Programmverifikation

- Zusicherungen $--\#$ *assert*
- Vor- und Nachbedingungen werden mit $--\#$ *pre* bzw. $--\#$ *post* deklariert.
- Beispiel:

```
procedure Increment (X : in out Counter_Type);
--# derives X from X;
--# pre X < Counter_Type'Last;
--# post X = X~ + 1;
```

SPARK Toolset

- Der *Examiner* generiert *verification conditions* (VCs).

SPARK Toolset

- Der *Examiner* generiert *verification conditions* (VCs).
- Der *Simplifier* vereinfacht die VCs. Wenn die VCs zu **true** reduziert werden können, dann entspricht das Programm der annotierten Spezifikation.

Beispiel

```
type T is range -128 .. 128;

procedure Inc (X : in out T)
--# derives X from X;
is
begin
   X := X + 1;
end Inc;
```

Beispiel

```
procedure_inc_1.
H1:   true .
H2:   x >= t_first .
H3:   x <= t_last .
       ->
C1:   x + 1 >= t_first .
C2:   x + 1 <= t_last .
```

Beispiel

```
procedure_inc_1.
H1:   x >= -128  .
H2:   x <= 128   .
        ->
C1:   x <= 127   .
```

Beispiel

```
procedure_inc_1.
H1:   x >= -128  .
H2:   x <= 128   .
        ->
C1:   x <= 127   .
```

Der Benutzer kann VCs mit dem *Proof Checker* beweisen

Examiner

Examiner

- Syntaktische Analyse

Examiner

- Syntaktische Analyse
- Analyse der Programm Struktur

Examiner

- Syntaktische Analyse
- Analyse der Programm Struktur
- Semantische Analyse

Examiner

- Syntaktische Analyse
- Analyse der Programm Struktur
- Semantische Analyse
- Daten- und Informationsflussanalyse:

Daten- und Informationsflussanalyse

- Referenz auf nichtinitialisierte Variablen

Daten- und Informationsflussanalyse

- Referenz auf nichtinitialisierte Variablen
- Ineffektive Anweisung

Daten- und Informationsflussanalyse

- Referenz auf nichtinitialisierte Variablen
- Ineffektive Anweisung
- Inkonsistenzen im Informationsfluss

Daten- und Informationsflussanalyse

- Referenz auf nichtinitialisierte Variablen
- Ineffektive Anweisung
- Inkonsistenzen im Informationsfluss
- Schleifen Stabilität

Daten- und Informationsflussanalyse

- Referenz auf nichtinitialisierte Variablen
- Ineffektive Anweisung
- Inkonsistenzen im Informationsfluss
- Schleifen Stabilität
- Nichtbenutzte Variablen

Daten- und Informationsflussanalyse

- Referenz auf nichtinitialisierte Variablen
- Ineffektive Anweisung
- Inkonsistenzen im Informationsfluss
- Schleifen Stabilität
- Nichtbenutzte Variablen
- Variablen, deren Wert konstant bleibt

Laufzeitfehleranalyse

- Array Index nicht im Bereich

Laufzeitfehleranalyse

- Array Index nicht im Bereich
- Wertbereichfehler

Laufzeitfehleranalyse

- Array Index nicht im Bereich
- Wertbereichfehler
- Division durch 0

Laufzeitfehleranalyse

- Array Index nicht im Bereich
- Wertbereichfehler
- Division durch 0
- Numerischer Überlauf

Laufzeitfehleranalyse

- Der *Run-Time Error Checker* generiert systematisch für alle möglichen Laufzeitfehler Bedingungen, unter denen diese nicht auftreten.

Laufzeitfehleranalyse

- Der *Run-Time Error Checker* generiert systematisch für alle möglichen Laufzeitfehler Bedingungen, unter denen diese nicht auftreten.
- Der *Simplifier* kann viele dieser Bedingungen erfüllen.

Laufzeitfehleranalyse

- Der *Run-Time Error Checker* generiert systematisch für alle möglichen Laufzeitfehler Bedingungen, unter denen diese nicht auftreten.
- Der *Simplifier* kann viele dieser Bedingungen erfüllen.
- Die übrig gebliebenen Bedingungen können mit dem *Proof Checker* erfüllt werden.

Programmverifikation

- Partielle Korrektheit:
  Vorbedingung ist erfüllt
  Programm terminiert
  Nachbedingung ist erfüllt

Programmverifikation

- Partielle Korrektheit:
  Vorbedingung ist erfüllt
  Programm terminiert
  Nachbedingung ist erfüllt
- Totale Korrekheit:
  Programm ist partiell korrekt und terminiert immer

Programmverifikation

- Partielle Korrektheit:
  Vorbedingung ist erfüllt
  Programm terminiert
  Nachbedingung ist erfüllt
- Totale Korrekheit:
  Programm ist partiell korrekt und terminiert immer
- Die Korrektheit wird mit dem SPARK Toolset überprüft.

Programmverifikation

- Partielle Korrektheit:
  Vorbedingung ist erfüllt
  Programm terminiert
  Nachbedingung ist erfüllt
- Totale Korrekheit:
  Programm ist partiell korrekt und terminiert immer
- Die Korrektheit wird mit dem SPARK Toolset überprüft.
- Worst-Case Ausführungszeit wird ermittelt.

Spracheigenschaften

- Keine Exceptions

Spracheigenschaften

- Keine Exceptions
- Keine Rekursion

Spracheigenschaften

- Keine Exceptions
- Keine Rekursion
- Keine dynamische Speicherverwaltung

Spracheigenschaften

- Keine Exceptions
- Keine Rekursion
- Keine dynamische Speicherverwaltung
- Keine generischen Typen

Spracheigenschaften

- Keine Exceptions
- Keine Rekursion
- Keine dynamische Speicherverwaltung
- Keine generischen Typen
- Funktionen dürfen keine Seiteneffekte enthalten

Objektorientierung

- Konstruktoren und Destruktoren $\Rightarrow$ Explizite Speicherverwaltung

Objektorientierung

- Konstruktoren und Destruktoren $\Rightarrow$ Explizite Speicherverwaltung
- Kapselung: **package** als Klasse

Objektorientierung

- Konstruktoren und Destruktoren $\Rightarrow$ Explizite Speicherverwaltung
- Kapselung: **package** als Klasse
- Abstraktion mit der Deklaration von **package**s

Objektorientierung

- Konstruktoren und Destruktoren $\Rightarrow$ Explizite Speicherverwaltung
- Kapselung: **package** als Klasse
- Abstraktion mit der Deklaration von **package**s
- Modulare Programmierung

Objektorientierung

- Konstruktoren und Destruktoren $\Rightarrow$ Explizite Speicherverwaltung
- Kapselung: **package** als Klasse
- Abstraktion mit der Deklaration von **package**s
- Modulare Programmierung
- Vererbung

Objektorientierung

- Konstruktoren und Destruktoren $\Rightarrow$ Explizite Speicherverwaltung
- Kapselung: **package** als Klasse
- Abstraktion mit der Deklaration von **package**s
- Modulare Programmierung
- Vererbung
- Keine Polymorphie

Beispiel

```
package P is
   procedure Exchange(X, Y : in out Integer);
   --# derives X from Y &
   --#             Y from X;
   --# post X = Y~ and Y = X~;
end P;

package body P is
   procedure Exchange(X, Y : in out Integer)
   is
      T : Integer;
   begin
      T := X; X := Y; Y := T;
   end Exchange;
end P;
```

Beispiel

```
procedure_exchange_1.
H1:     true  .
H2:     x >= integer_first  .
H3:     x <= integer_last  .
H4:     y >= integer_first  .
H5:     y <= integer_last  .
         ->
C1:     y = y  .
C2:     x = x  .
```

Zusammenfassung

- SPARK analysiert Programme zur Entwicklungszeit.

Zusammenfassung

- SPARK analysiert Programme zur Entwicklungszeit.
- Das fertige Programm verhält sich garantiert den annotierten Spezifikationen entsprechend.

Zusammenfassung

- SPARK analysiert Programme zur Entwicklungszeit.
- Das fertige Programm verhält sich garantiert den annotierten Spezifikationen entsprechend.
- Mit SPARK entwickelte Programme genügen strengen Standards aus Industrie und Militär.