Synchronizing Resources

Seminar: Programmiersprachenkonzepte

15.04.2004

Tobias Blasche

Inhalt

- **Grundlegende Konzepte**
  - Datentypen
  - Kontrollfluss
  - Abstraktion/Komplexität
- **Verteilung, Nebenläufigkeit und Konsistenz**
  - Virtual Machine
  - Threads/Prozesse
  - Synchronisation und Kommunikation

Inhalt

- **Grundlegende Konzepte**
  - **Datentypen**
  - Kontrollfluss
  - Abstraktion/Komplexität
- **Verteilung, Nebenläufigkeit und Konsistenz**
  - Virtual Machine
  - Threads/Prozesse
  - Synchronisation und Kommunikation

Datentypen

- **elementare Datentypen**
  - integer, boolean, real, character, string
- **strukturierte Datentypen**
  - array, record, union
- **benutzerdefinierte Datentypen**
  - enumeration, record, union, pointer
  - Deklaration: `type` *type_id* = *type_definition*

Inhalt

- **Grundlegende Konzepte**
  - Datentypen
  - **Kontrollfluss**
  - Abstraktion/Komplexität
- **Verteilung, Nebenläufigkeit und Konsistenz**
  - Virtual Machine
  - Threads/Prozesse
  - Synchronisation und Kommunikation

Kontrollfluss

- **Iterativ**

  Do Anweisung:

  – `do` *guarded_command* `[]` *guarded_command* `[] ... od`

  – unendliche Interation

  For-All Anweisung:

  – `fa` *quantifier, quantifier, ... ->* `af`

  – endliche Iteration

- **Selektiv**

  If Anweisung:

  – `if` *guarded_command* `[]` *guarded_command* `[] ... fi`

  – Kombination aus If und Case Anweisung anderer Sprachen

Inhalt

- **Grundlegende Konzepte**
  - Datentypen
  - Kontrollfluss
  - **Abstraktion/Komplexität**
- **Verteilung, Nebenläufigkeit und Konsistenz**
  - Virtual Machine
  - Threads/Prozesse
  - Synchronisation und Kommunikation

Abstraktion/Komplexität

- **Prozedur**
  - benannte Abfolge von Programmanweisungen
  - Deklaration und Implementation können getrennt werden
  - Parameterübergabe

    `val:` Werte werden „hineinkopiert“

    `res:` Werte werden „herauskopiert“

    `var:` Werte werden „hinein und herauskopiert“

    `ref:` Adressen werden übergeben
  - Aufruf

    `call:` synchroner Aufruf, lokale Umgebung

    `send:` asynchroner Aufruf, verteilte/nebenläufige Umgebung

Abstraktion/Komplexität

- **Prozedur – Beispiel**

Kurzform

```
 procedure summe(n: int) returns sum: int
      sum := 0
      fa i:= 1 to n -> sum +:= i af
 end
```

ausführlich

```
op summe(n: int) returns sum: int
proc summe(n: int) returns sum: int
      sum := 0
      fa i:= 1 to n -> sum +:= i af
end
```

Abstraktion/Komplexität

- **Resource**
  - Kapselt Prozeduren/Operation+Procs, Typen, Variablen
  - Bestandteile: `spec, body`
  - Spec
    - Deklariert Schnittstelle, z.B. Operationen, Konstanten
    - Implizites exportieren deklarierte Objekte
  - Body
    - Implementation der deklarierten Operationen
  - `spec` und `body` können separat programmiert werden
  - Erzeugen von Instanzen: `create` resource_name (args)
  - Zerstören von Instanzen: `destroy` *expr*
  - Importieren: `import` *name, name, ...*

Abstraktion/Komplexität

- **Resource – Beispiel**

```
resource Stack
  type result = enum(OK, OVERFLOW, UNDERFLOW)
  op push(item: int) returns r: result
  op pop(res item: int) returns r: result
body Stack(size: int) separate

body Stack
  # some code
end
```

Abstraktion/Komplexität

- **Global**
  - Menge von Objekten, verwendet von Resourcen und Globals
  - Eine Instanz pro Adressraum
  - Typen:
    - einfach, komplex
  - Einfach
    - Deklaration von Konstanten, Typen und Variablen
  - Komplex
    - `spec` und `body`
    - Zusätzlich Deklaration von Operationen möglich
    - Implementation dieser Operationen
    - Möglichkeit der Konstruktion von Bibliotheken

Zusammenfassung

- Grundlegende Konstrukte
  - Datentypen,
  - Kontrollstrukturen
- Abstraktionen
  - Prozedur/(Operation + Proc)
    - Kapselt Algorithmus
  - Resource
    - Kapselt Operationen, Variablen und Konstanten
  - Global
    - Kapselt global verwendete Objekte
  - Capabilities
    - Pointer auf Operationen, Resourcen und Virtual Machines

Inhalt

- **Grundlegende Konzepte**
  - Datentypen
  - Kontrollfluss
  - Abstraktion/Komplexität
- **Verteilung, Nebenläufigkeit und Konsistenz**
  - **Virtual Machine**
  - Threads/Prozesse
  - Synchronisation und Kommunikation

Virtual Machine

- Definiert einen Adressraum
- Unterstützt echte Verteilung von SR Programmen
- Erzeugung
  - Dynamisch
  - `create vm() on` *expr*
- Inhalt
  - globals, resources
  - Erzeugung: `create` *res_name (args)* `on` *expr*
- Referenzierung
  - durch Capabilitiy vom Typ `vm`

Inhalt

- **Grundlegende Konzepte**
  - Datentypen
  - Kontrollfluss
  - Abstraktion/Komplexität
- **Verteilung, Nebenläufigkeit und Konsistenz**
  - Virtual Machine
  - **Prozesse/Threads**
  - Synchronisation und Kommunikation

Prozesse/Threads

- Analogon zu Prozeduren
- Deklaration

`process` *process_id (quantifier, quantifier, ...)*
block
`end`

`op` *process_id (quantifier, quantifier, ...)* `{send}`
`proc` *process_id (quantifier, quantifier, ...)*
block
`end`

Prozesse/Threads

- Erzeugung

  Co-Begin

  - `co` concurrent_command // concurrent_command // ... `oc`
  - erzeugen eines od. mehrerer Prozesse
  - Code in verschiedenen Prozessen kann verschieden sein
  - Bsp.: `co p() // q() oc`

  Parallel loops

  - `co` (quantifier) concurrent_command `oc`
  - Mehrfacherzeugung durch quantifizierenden Ausdruck
  - Code in verschiedenen Prozessen gleich
  - Daten unterschiedlich
  - Bsp.: `co (i:=1 to 4) r(i) oc`

Prozesse/Threads

- Erzeugung

  launch-at-elaboration
  - Erzeugung bei Abarbeitung einer `process` Deklaration
  - Imlizit bei Instanziierung der umgebenden `resource`
  - Code in verschiedenen Prozessen kann unterschiedlich sein
  - Bsp.: siehe Deklaration 1. Möglichkeit

  Fork/Join
  - Erzeugung ist explizit ausführbare Anweisung
  - „`send` style forks“
  - Bsp.: `send foo()`
  - kein explizites join
  - dafür aber andere Ausdrucksmöglichkeiten der Synchronisation

Inhalt

- **Grundlegende Konzepte**
  - Datentypen
  - Kontrollfluss
  - Abstraktion/Komplexität
- **Verteilung, Nebenläufigkeit und Konsistenz**
  - Virtual Machine
  - Threads/Prozesse
  - **Synchronisation und Kommunikation**

Synchronisation

- Semaphore
  - „low level“ Synchronisation für „shared data“
  - `sem` *sem_definition, sem_definition, ...*
  - Standard Operationen: `P` (*passeren),* `V` (*vrygeven*)
  - Warteschlangetyp für Operation `P`: FIFO

  Beispiel

```
resource cs()
  ...          # define N and x
  sem mutex := 1
  process p(i := 1 to N)
    ...        # non-critical section
    P(mutex) # enter critical section
    ...
    V(mutex) # exit critical section
  end
end
```

Kommunikation

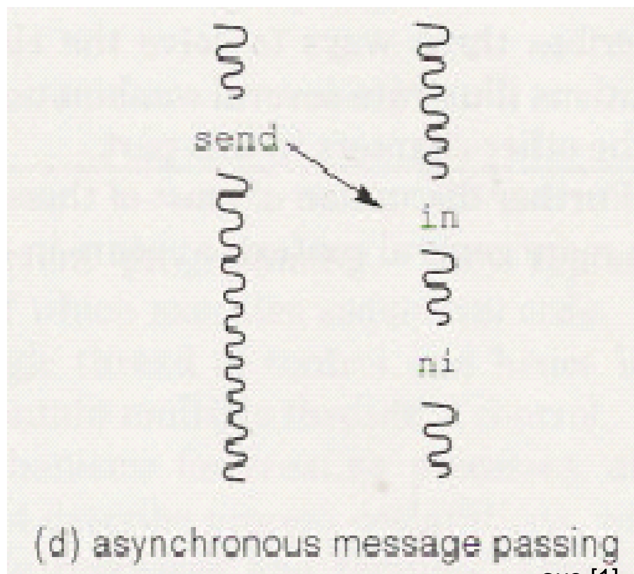
- basiert auf verschiedener Anwendung einer Operation

Invoke	*Service*	*Effect*
`call`	`proc`	(possibly remote) procedure call
`call`	`in`	rendezvous
`send`	`proc`	dynamic process creation
`send`	`in`	asynchronous message passing

aus [2] Seite 3

Kommunikation

- asynchrones Message Passing
  - Operationsdeklaration besitzt keinen entsprechenden `proc` Teil
  - Deklaration einer `receive` Anweisung innerhalb eines Prozesses
  - `receive` *op_id subscripts (variable, variable, ...)*
  - `receive` definert eine FIFO – Queue für Operationen „*op_id*“
  - `receive` blockiert ausführenden Prozess bis Aufruf „op_id“

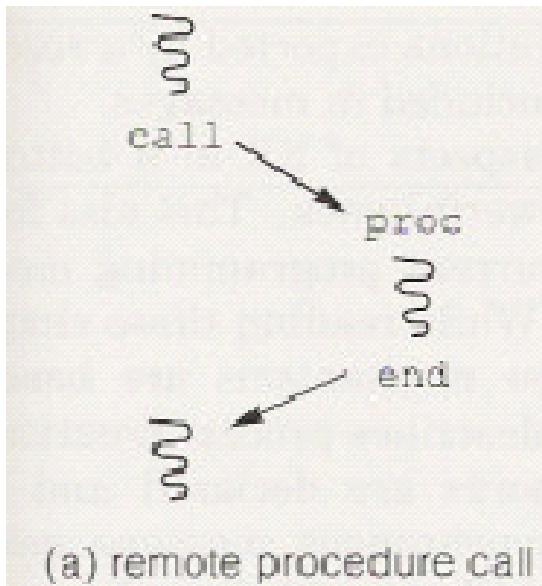


(d) asynchronous message passing

aus [1]

Kommunikation

- Remote Procedure Call
  - syntaktisch wie lokaler Prozeduraufruf
  - somit transparent für den Benutzer
  - logischerweise jedoch länger andauerend



(a) remote procedure call

aus [1]

Kommunikation

- Rendezvous
  - Aufruf wird durch einen existierenden Prozess bearbeitet
  - Operationsdeklaration besitzt keinen entsprechenden `proc` Teil
  - Deklaration einer `in` Anweisung innerhalb eines Prozesses
  - `in` blockiert ausführenden Prozess bis Aufruf

`in` ***op_command [] op_command[] ...*** **`ni`**

  - `in` definert eine FIFO – Queue für Operationen
  - *op_command* spezifiziert eine Operation
  - Selektion aus FIFO – Queue möglich, durch `st` und `by`

operation (formal_id_list) **`returns`** ***result_id***

`st` ***synch_expr*** **`by`** ***sched_expr -> block***

Kommunikation

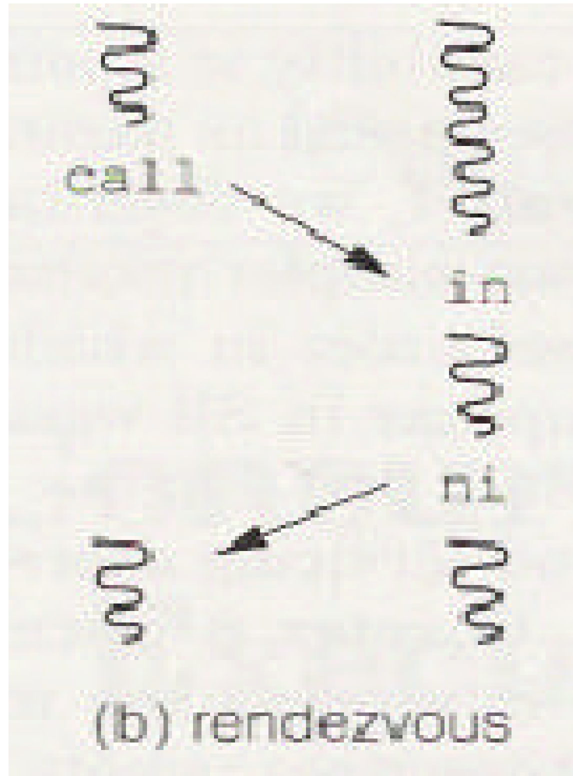
- Rendezvous

  `st` (such that)
  - Selektion aus den wartenden Operationsaufrufen
  - Referenzierung von Operationsparametern möglich
  - Beispiel: `in a(x) st x=3 -> ... ni`

  `by`
  - Selektion aus Vorauswahl von `st`
  - Minimal-Kriterium: kleinster Wert wird bevorzugt
  - Beispiel: `in a(x) st x<3 by x -> ... ni`

Kommunikation

- Rendezvous



aus [1]

Literatur

- [1] The SR Programming Language: Concurrency in Practice
  Gregory R. Andrews, Ronald A. Olsson
  The Benjamin/Cummings Publishing Company Inc., 1993
- [2] SR: A Language for Parallel and Distributed Programming
  G. R. Andrews, R. A. Olsson, Michael H. Coffing, Gregg M. Townsend
  http://www.cs.arizona.edu/sr/language.pdf
- [3] Programming Language Pragmatics
  Michael L. Scott
  Morgan Kaufmann Publishers, San Francisco California, 2000