SEMINAR "PROGRAMMIERUNG VERTEILTER SYSTEME"

TOOLS FÜR ERLANG: CODEGENERATOREN, DEBUGGER, TESTUMGEBUNGEN UND PROFILER

Magnus Stiller

Betreuer: Frank Huch

Literatur:

- Joe Armstrong, Robert Virdings, Claes Wikström, Mike Williams (1996), "Concurrent Programming in ERLANG – Second Edition", Prentice Hall
- "Erlang/OTP R9B Documentation", http://www.erlang.org/doc/r9b/doc/index.html

Inhaltsverzeichnis

1 Einleitung

Spätestens seit der Etablierung des World Wide Webs als weltumspannendes Informationsmedium ist es für die meisten Menschen offensichtlich geworden, dass wir in einer Welt leben, in der fast alle Computer über verschiedene Netzwerke miteinander verbunden sind. Während in der Vergangenheit auf den lokalen und globalen Netzwerken zuerst nur Dateien und Nachrichten transportiert worden sind, wurde im Laufe der Zeit auch Anwendungen entwickelt, die auf mehreren Computern arbeiten.

Um diese Anwendungen programmieren zu können, werden an die benutzten Programmiersprache besondere Anforderungen gestellt, wie zum Beispiel das Starten von Prozessen auf anderen Computer. Um diesen Anforderungen gerecht zu werden, wurden bestehende Programmiersprache um Konzepte für Verteilte Systeme erweitert. Ein anderer Weg diesen Anforderungen gerecht zu werden, ist die Entwicklung von neuen Programmiersprachen.

Eine dieser neuen Programmiersprachen ist Erlang. Die ungetypte funktionale Programmiersprache ist nach dem dänischen Mathematiker Agner Krarup Erlang (1878 – 1929) benannt und wurde in den Ericsson and Ellemtel Computer Science Laboratories entwickelt.

Das Konzept von Erlang macht keinen Unterschied zwischen der Programmierung von nebenläufigen und Verteilten Systemen. Beim Aufruf von Erlang wird ein Erlang Knoten gestartet. Von diesem Knoten aus kann eine Anwendung einen Prozess sowohl auf einem lokalen als auch auf einem anderen Knoten starten, wobei der andere Knoten auf demselben oder auf einem anderen Computer liegen kann. Der neue Prozess hat einen eindeutigen Process Identifier (Pid) und ist deshalb von allen anderen Prozessen, die seine Pid kennen, ansprechbar. Dazu schickt ein Prozess eine Nachricht mittels der Pid des gewünschten Prozesses an dessen Mailbox. Diese Mailbox ist durch die Pid des Prozesses eindeutig identifizierbar. Aus der Mailbox kann sich der Prozess nun durch Pattern Matching eine Nachricht heraussuchen und abarbeiten.

Diese Konzepte machen in Erlang eine elegante Programmierung nebenläufiger und Verteilter Systeme möglich.

Wichtig für die Entwicklung von Anwendungen in Verteilten Systemen sind aber neben der Programmiersprache auch die Werkzeuge. Im besten Fall müssten sie in der Lage sein, eine Anwendung auf allen beteiligten Computern zu überwachen und die eventuell auftretenden Fehler (wie zum Beispiel Deadlocks) zu finden und zu verstehen.

Erlang stellt in der Open Telecom Platform (OTP) Library eine Anzahl von Werkzeugen bereit, und diese Ausarbeitung soll zeigen, ob sie die Anforderungen zur Entwicklung von Anwendungen in Verteilten Systemen erfüllen. Bei den nachfolgenden Tools sind zur Reduzierung des Umfangs der Ausarbeitung nur ausgewählte Optionen und Funktionen erläutert.

2 Compiler und seine Tools

Der Erlang Compiler verwandelt korrekten Erlang Quellcode in Bytecode, ansonsten wird eine Fehlermeldung ausgegeben. Der Compiler ist im Modul `compiler` und kann wie folgt aufgerufen werden: `compile:file(File, Flags).`

Zur Vereinfachung des Aufrufs gibt es folgende Möglichkeiten:

Vereinfachung	Äquivalent zu
`compile:file(File)`	`compile:file(File, [verbose,report])`
`c(File)`	`compile:file(File, [report])`
`c(File, Flags)`	`compile:file(File, Flags ++ [report])`

Zu den für die Seminararbeit interessanten Flags gehören folgende:
- `debug_info` ist notwendig, um den Debugger nutzen zu können. Allerdings kann dann der Quellcode aus dem Bytecode erzeugt werden.
- `'P'` erzeugt ein Listing des Quellcodes, nachdem der Preprozessor ausgeführt wurde und der Quellcode geparst wurde. Im neu erzeugten Quellcode steht immer nur ein Ausdruck pro Zeile, so dass er zur Benutzung des Debuggers und anderer Tools besser lesbar ist. Allerdings werden die Kommentare entfernt.

Zum Compilieren von größeren Projekten ist es hilfreich, das Modul `make` zu benuten. Analog zum Unix Befehl `make` werden alle Dateien im aktuellen Verzeichnis compiliert,
- deren Objektdatei nicht existiert oder
- deren Quellcode seit dem letzten Compilieren verändert wurde oder
- deren Include-Dateien seit dem letzten Compilieren der Datei verändert wurde.

Der Aufruft erfolgt über `make:all()` oder über `make:all(Options)`, wobei Options eine Liste von Optionen ist, die auch die Optionen des Compilers sein können. Der Rückgabewert ist entweder `up_to_date` oder `error` im Fehlerfall.

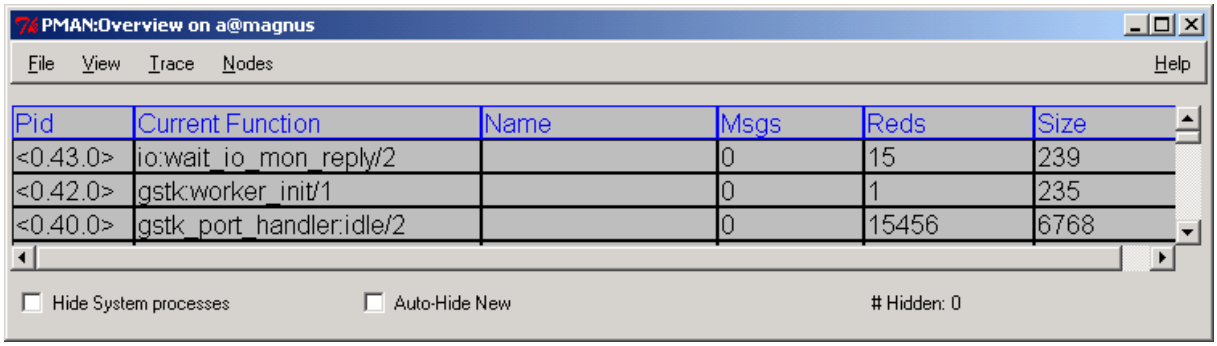
Zu den im Rahmen der Seminararbeit interessanten Optionen gehören:
- `noexec` gibt an, dass nur angezeigt werden soll, ob eine Datei seit ihrer letzten Compilierung verändert worden ist. Es wird keine Objektdatei erzeugt.
- `netload` lädt alle neu compilierten Module in alle bekannten Knoten. Wenn zum Beispiel ein Datenbankserver über Klienten abgefragt wird, die auf verschiedenen Rechnern laufen, lässt sich mittels `netload` eine Änderung im Programmcode des Klienten allen Rechner vermitteln, ohne dass am jeweiligen Rechner des Klienten Wartungsarbeit anfällt.

`make:all()` überprüft alle Dateien im aktuellen Verzeichnis. Alternativ kann auch eine Datei namens `Emakefile` angelegt werden, so dass nur die dort spezifizierten Dateien bearbeitet werden, z.B.:

```
test.
'../counter/server'.
```

3 Prozessmanager Pman

Der Prozessmanager Pman zeigt lokale oder auf bekannten Knoten laufende Prozesse an.



Der Prozessmanager wird im Modul `pman` über `pman:start()` aufgerufen.

In der Tabelle werden folgende Informationen zu einem Prozess angezeigt:
- Pid: der Process Identifier
- Current Function: die gerade ausgeführte Funktion (Modul:Funktion/Stelligkeit)
- Name: der registrierte Name, wenn vorhanden
- Msgs: Anzahl der Nachrichten in der Mailbox
- Reds: Anzahl der Reduktionen seit Prozessstart
- Size: ungefährer benutzer Speicher berechnet aus der Größe des Stacks und des Heaps

Unterhalb der Tabelle kann man die Anzeige von Systemprozessen und neuen Prozessen beeinflussen. Außerdem wird die Anzahl der nicht angezeigten Systemprozesse angezeigt.

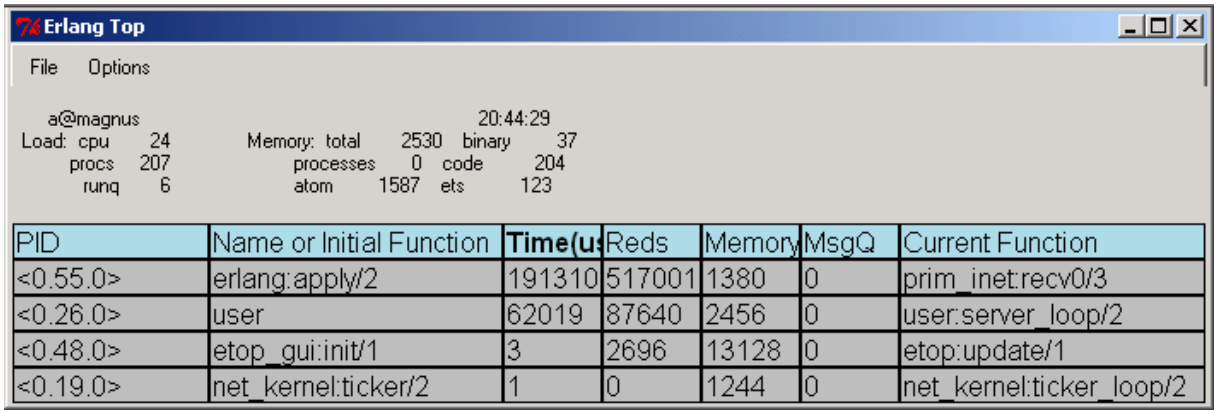
Zum Konfigurieren des Prozessmanagers kann man im Menü File Optionen einstellen und abspeichern. Die Anzeige von Modulen und Prozessen lässt sich im Menü View detailliert einstellen. Außerdem lassen sich dort weitere Informationen zum ausgewählten Modul anzeigen. Im Menü Trace lässt sich der selektierte Prozess beenden oder das Trace Fenster für ihn öffnen.

Im Menü Nodes wird der Knoten ausgewählt, dessen Prozesse angezeigt werden sollen. Standardmäßig ist der lokale Knoten angezeigt, aber wenn andere Knoten bekannt sind, können sie ebenfalls ausgewählt werden.

Um genauere Informationen zu einem Prozess zu erhalten, muss in der Tabelle die entsprechende Zeile des Prozess mit Doppelklick markiert werden. Die dort erhaltenen Informationen werden in Kapitel 7.4 näher erläutert.

4 Erlang Top

Erlang Top (etop) zeigt Informationen über laufende Prozesse an – ähnlich präsentiert wie bei dem Unixprogramm top.



Zuerst gibt Erlang Top Informationen über den Erlang Knoten aus: Name und Host des Knotens, Prozessorauslastung, Anzahl der vorhandenen und der lauffähigen Prozesse und welcher Speicher wofür benutzt wird (in Kilobytes).
Dann folgt eine sortierte Tabelle mit Informationen zu den Prozessen, die der Tabelle des Prozessmanager Pman sehr ähnelt. Zusätzlich gibt es für jeden Prozess noch Angaben über seine Laufzeit und die Anzahl der Nachrichten in der Mailbox.

Im Menü Options kann eingestellt werden, nach welchem Kriterium die Tabelle sortiert wird, in welchen Intervallen die Anzeige aktualisiert wird und wie viele Prozesse angezeigt werden.

Erlang Top wird aus dem Modul `etop` über `etop:start(Optionen)` aufgerufen. Eine notwendige Option ist die Angabe des zu überwachenden Knotens mittels `{node, a@host}`. So lässt sich auch die Auslastung von anderen Knoten anzeigen. Andere Optionen sind dieselben, die man auch im Menü Options bearbeiten kann. Falls statt einer graphischen Ausgabe eine Textausgabe erwünscht ist, wird das durch die Option `{output, text}` erreicht.

5 Debugger

Der Debugger ermöglicht die Fehlersuche in Erlang Programmen. Unter anderem kann man den Programmcode schrittweise durchgehen, gezielt mit einem Breakpoint anhalten oder den Wert von Variablen auslesen oder verändern. Dazu muss aber der zu debuggende Quellcode mit der Option `debug_info` compiliert werden.

5.1 Testmodul

Zum Testen des Debugger (und auch der Tracer) sind Programme interessant, die im Laufe der Zeit in einen Fehlerzustand geraten. Als Beispiel ist hier das bekannte Problem der dinierenden Philosophen gewählt. Das Problem ist so implementiert, dass es einen Deadlock gibt, wenn alle Philosophen den linken Stick gegriffen haben.
Sowohl für den Debugger (als auch für die späteren Tools) sind besonders die ausführbaren Zeilen wichtig. Eine ausführbare Zeile ist eine Zeile, die einen ausführbaren Ausdruck enthält (wie z.B. eine Funktion). Zur Anschauung sind daher ausführbaren Zeilen mit der Zeilennummer versehen.

```
-module(dining).
-export([start/3,stickDown/0,startPhil/3]).

start(N,NodeStick,NodesPhils) ->
    Sticks = createSticks(N,NodeStick),                    % 5
    createPhils(lists:last(Sticks),Sticks,0,NodesPhils).  % 6
```

Zum Starten des Programms muss die Anzahl der Philosophen und die Namen der Knoten angegeben werden, auf denen das Programm laufen soll. `NodeStick` bestimmt den Knoten, auf dem die Prozesse der Sticks laufen sollen, und `NodesPhils` ist eine Liste von Knoten, auf dem die Philosophen arbeiten sollen. Die Anzahl der Knoten in der Liste muss mindestens der Anzahl der Philosophen entsprechen. Wenn die Namen der Knoten alle dem lokalen Computer entsprechen, läuft das Programm nur auf einem Computer.

```
createSticks(0,_) ->
    [];                                                              % 9
createSticks(N,NodeStick) ->
   [spawn(NodeStick,dining,stickDown,[])|createSticks(N - 1,NodeStick)].%11
```

Jeder Stick wird durch einen Prozess implementiert.

```
createPhils(SL,[],_,_) ->
    ok;                                                % 14
createPhils(SL,[SR|Sticks],N,[NodePhil|NodesPhils]) ->
    spawn(NodePhil,dining,startPhil,[SL,SR,N]),        % 16
    createPhils(SR,Sticks,N + 1,NodesPhils).           % 17
```

Jedem Philosoph, der durch einen Prozess implementiert wird, wird ein linker und ein rechter Stick zugewiesen.

```
stickDown() ->
    receive             % 20
        {take,P} ->
            P ! took,   % 22
            stickUp()   % 23
    end.

stickUp() ->
    receive             % 27
        put ->
            stickDown() % 29
    end.
```

Der Prozess eines Sticks kann entweder die Funktion `stickUp` oder `stickDown` durchlaufen. Dadurch wird auch gleichzeitig der Zustand des Sticks beschrieben. Der Zustand ändert sich, wenn der Prozess eine passende Nachricht eines Prozesses eines Philosophen erhalten hat.

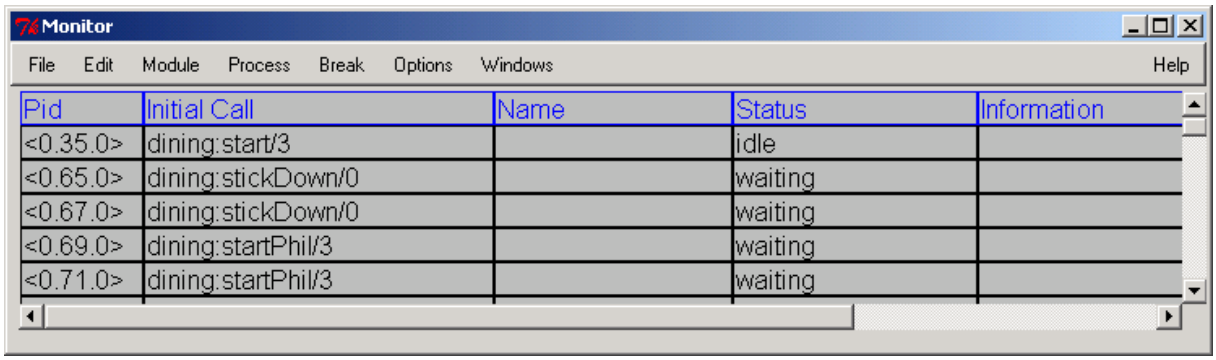
```
startPhil(SL,SR,N) ->
    random:seed(N*element(3,now())*10,N,element(3,now())), % 33
    phil(SL,SR,N).                                         % 34

phil(SL,SR,N) ->
    timer:sleep(random:uniform(100)),        % 37
    io:format("~w is thinking~n",[N]),       % 38
    io:format("~w is getting hungry~n",[N]), % 39
    SL ! {take,self()},                      % 40
    receive                                  % 41
        took ->
            ok                               % 43
    end,
    io:format("~w took left stick~n",[N]),   % 45
    SR ! {take,self()},                      % 46
    receive                                  % 47
        took ->
            ok                               % 49
    end,
    io:format("~w took right stick~n",[N]),  % 51
    io:format("~w is eating~n",[N]),         % 52
    timer:sleep(random:uniform(100)),        % 53
    SL ! put,                                % 54
    SR ! put,                                % 55
    io:format("~w put both sticks~n",[N]),   % 56
    phil(SL,SR,N).                           % 57
```

Nachdem ein Zufallsgenerator initialisiert worden ist, macht der Prozess des Philosophen immer das gleiche: Eine zufällige Zeit lang denken, dann den linken Stick nehmen, danach den rechten Stick nehmen, eine zufällige Zeit lang essen und dann beide Sticks wieder hinlegen.
Zum Nehmen eines Sticks schickt der Prozess eine Nachricht an den passenden Prozess eines Sticks. Sobald er ihn nehmen kann, bekommt er eine Nachricht vom passenden Prozess.

5.2 Monitor Fenster

Der Debugger wird aus dem Modul `debugger` mittels `debugger:start()` aufgerufen. Es wird das Monitor Fenster geöffnet, das Informationen über alle zu debuggende Prozesse enthält.



In der Tabelle erscheinen folgende Informationen zu den überwachten Prozessen:

- Pid: der Process Identifier
- Initial Call: die Funktion (Modul:Funktion/Stelligkeit), die der Prozess zum Starten aufgerufen hat
- Name: der registrierte Name, wenn vorhanden
- Status: der Prozess kann sechs Zustände annehmen:
  - idle: Die aufgerufene Funktion ist abgearbeitet.
  - running: Der Prozess wird bearbeitet.
  - waiting: Der Prozess wartet auf ein Nachricht.
  - break: Der Prozess wurde an einem Breakpunkt angehalten.
  - exit: Der Prozess wurde beendet.
  - no_conn: Die Verbindung zum Prozess auf einem anderen Knoten ist abgebrochen.
- Information: zusätzliche Angaben bei folgenden Zuständen
  - exit: Grund für die Terminierung
  - break: Tupel mit Modul und Zeile, an dem der Prozess angehalten wurde

Das obige Screenshot ist während des Durchlaufs des Testmoduls mit zwei Philosophen entstanden. Beide Philosophen warten auf den Erhalt einer Nachricht. Da auch die Prozesse der Sticks auf eine Nachricht warten, ist ein Deadlock entstanden.

Im Menü Edit kann man aus der Tabelle alle terminierten Prozesse entfernen oder einen ausgewählten Prozess terminieren lassen.

Im Menü Module lassen sich die Module einstellen, die vom Debugger überwacht werden. Über den Menüpunkt Interpret gelangt man in einen Dialog zur Auswahl eines zu debuggenden Moduls. Es können nur Module ausgewählt werden, die mittels der Option `debug_info` compiliert worden sind. Im Menü Module lassen sich die Module auch wieder aus dem Debugger entfernen.

5.3 Breakpoints

Sowohl im Monitor Fenster als auch im später erläuterten Attach Prozess Fenster lassen sich Breakpoints setzen, die einen Prozess anhalten. Ein Breakpoint kann aktiv oder inaktiv sein, wobei er im inaktiven Zustand ignoriert wird. Für jeden Breakpunkt kann man festlegen, was mit ihm geschieht, nachdem er erreicht und der Prozess angehalten worden ist:

- Enable : Der Breakpoint bleibt aktiv.
- Disable : Der Breakpoint wird inaktiv.
- Delete : Der Breakpoint wird gelöscht.

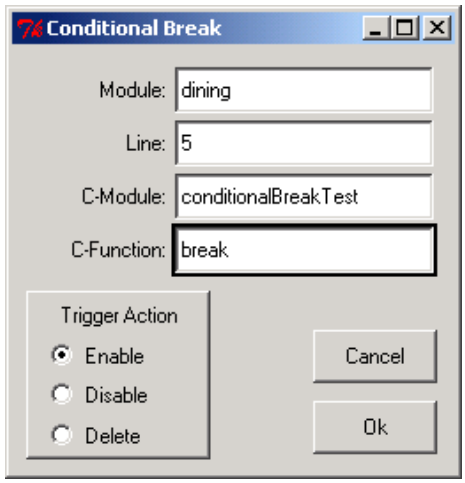
Es gibt drei Arten von Breakpoints (Line Breakpoint, Function Breakpoint und Conditional Breakpoint), die über Dialogfester konfiguriert werden können:

- Beim Line Breakpoint wird der Prozess in einem Modul an einer ganz bestimmten Zeile angehalten.
- Beim Function Breakpoint wird der Prozess beim Aufruf einer ganz bestimmten Funktion angehalten.
- Beim Conditional Breakpoint wird der Prozess an der gewünschten Zeile angehalten, wenn eine bestimmte Bedingung erfüllt wird. Diese Bedingung wird durch eine Funktion implementiert, die in einem beliebigen Modul programmiert worden sein kann.

Sobald der zu debuggende Prozess an der gewünschten Zeile angelangt ist, wird die zu testende Funktion aufgerufen. Sollte sie `true` zurückgeben, wird der Prozess angehalten, sonst nicht.

```
-module(conditionalBreakTest).
-export([break/1]).

break(Bindings) ->
    case int:get_binding('N', Bindings) of
        {value, N} ->
            N < 2;
        _ ->
            false
    end.
```

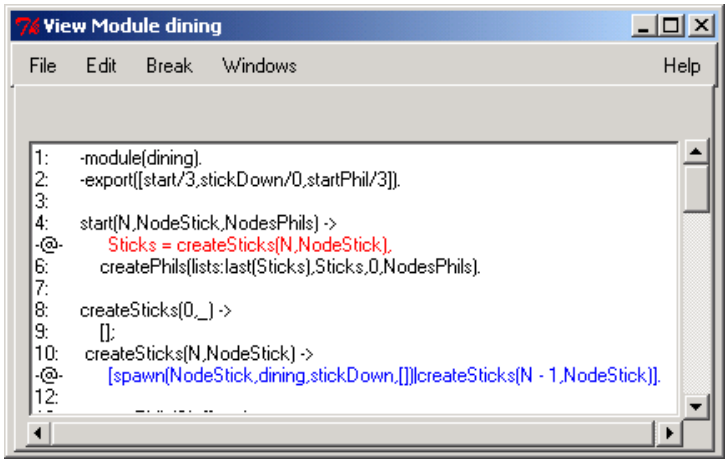


Im Beispiel hält der Computer den Prozess in der fünften Zeile an, wenn in der Testfunktion `N<2` ist. Das wird durch die Funktion `int:get_binding(Var,Bindings)`und ihrem Rückgabewert `{value,Value}` möglich. Da die Variable `Value` den Wert von `Var` (im Beispiel `N`) enthält, lassen sich Tests implementieren.

Beim Line Breakpoint und dem Conditional Breakpoint muss die spezifizierte Zeile eine ausführbare Zeile sein.

5.4 View Module Fenster

Da zum Setzen der Breakpoints immer eine Zeilennummer einer ausführbaren Zeile notwendig ist, gibt es zur einfacheren Einrichtung der Breakpoints das View Module Fenster. Es lässt sich über das Menü Module im Monitor Fenster aufrufen.

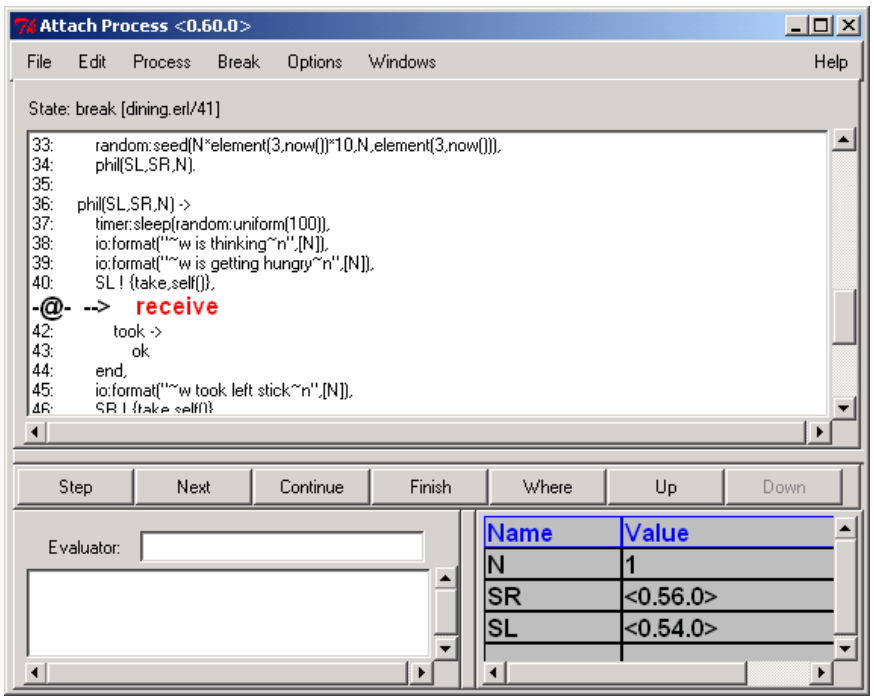


Der Quellcode wird inklusive der jeweiligen Zeilennummer angezeigt. Ein Breakpoint wird durch -@- angezeigt, wobei eine rote Schriftfarbe bedeutet, dass der Breakpoint aktiv ist. Eine blaue Schriftfarbe zeigt an, dass der Breakpoint inaktiv ist.

Breakpoints lassen sich über das Menü Break einrichten oder durch einen Doppelklick auf die gewünschte Zeile. Sollte diese Zeile schon einen Breakpoint enthalten, wird dieser gelöscht.

5.5 Attach Process Fenster

Nach einem Doppelklick auf einen Prozess im Monitor Fenster öffnet sich das Attach Process Fenster. Dabei wird ein laufender Prozess immer angehalten.



Im Attach Process Fenster wird im oberen Bereich der Quellcode inkl. Zeilennummer dargestellt. Wenn der Prozess angehalten wurde, ist die entsprechende Zeile mit `-->` markiert und fett hervorgehoben. Die Zeile ist dann noch nicht ausgeführt worden. Ein Breakpoint wird mit -@- angezeigt, wobei aktive Breakpoints in rot und inaktive in blau dargestellt werden. Zeilen, die schon ausgeführt worden sind, die einen Breakpoint enthalten oder die zum Anhalten des Prozesses geführt haben, werden in einigen Erlang Versionen in einer größeren Schrift dargestellt.

In der rechten Hälfte des unteren Bereich des Attach Process Fensters wird der Wert der aktuell gebunden Variablen angezeigt. Durch einen Doppelklick auf eine Variable lässt sich der Wert ändern. In der linken Hälfte kann man ebenfalls Variablen neue Werte zuweisen, aber auch komplexere Anweisungen ausführen lassen. Ist zum Beispiel `N` eine gebundene Variable, so kann man ihr mittels `N = N * 2` einen neuen Wert zuweisen.

Der Ablauf eines Prozesses lässt sich über diverse Befehle steuern, die entweder im Menü Process stehen oder sich in einer Auswahl unterhalb des Quellcodes als Schaltflächen befinden.

- Next: Ausführen der aktuellen Zeile und Anhalten vor der nächsten Zeile in dieser Funktion bzw. in der Funktion, die die aktuelle Funktion aufgerufen hat
- Step: Ausführen der aktuellen Zeile und Anhalten vor der nächsten Zeile, die auch in der durch die aktuelle Zeile aufgerufenen Funktion liegen kann
- Continue: Prozess ohne Anhalten weiterlaufen lassen
- Finish: Prozess ohne Anhalten weiterlaufen lassen, bis die aktuelle Funktion einen Wert zurückgibt
- Skip: Überspringen der aktuelle Zeile und Anhalten vor der nächsten Zeile (sollte die Zeile die letzte in der aktuellen Funktion sein, wird `skipped` zurückgegeben)
- Time Out: Simulieren eines Time Outs in einer `receive...after` Anweisung
- Stop: Prozess anhalten, so dass man ihn weiter debuggen kann
- Kill: Prozess beenden

- Messages: Anzeigen der in der Mailbox befindlichen Nachrichten
- Where: der Quellcode im oberen Bereich wird so platziert, dass die aktuelle Zeile sichtbar ist

Außerdem wird durch Up der Zustand vor dem Aufruf der Funktion dargestellt, in der der Debugger angehalten hat. Da der Zustand aber nur optisch dargestellt wird, sind keine Änderungen z.B. am Wert der Variablen möglich. Up lässt sich rekursiv aufrufen, solange Daten auf dem Stackframe vorhanden sind. Durch Down wird wieder der Zustand hergestellt, der vor dem Befehl Up vorhanden war.

Eingeschränkt sind die Befehle auch im Menü Process im Monitor Fenster zu finden.

5.6 Debuggen in Verteilten Systemen

Das Debuggen von Prozessen, die auf anderen Knoten laufen, ist mit dem Debugger nur möglich, wenn ein lokaler Prozess einen neuen Prozess auf einem anderen Knoten mittels `spawn/4` startet. Dann kann dieser (und die von ihm gestarteten) wie ein lokaler Prozess über das Monitor Fenster ausgewählt und debuggt werden. Die Entwickler des Debuggers empfehlen aber immer nur auf einem Knoten zu debuggen, da es sonst zu inkonsistenten Zuständen kommen kann.

6 Profiler

Um die Leistungsfähigkeit von Programmen einschätzen zu können, werden Profiler eingesetzt. Diese können einerseits feststellen, wie häufig eine Programmzeile oder auch eine ganze Funktion aufgerufen worden ist, oder anderseits die Zeit messen, die zum Ablauf einer Funktion oder eines ganzen Programms nötig ist. So lassen sich dann Engpässe oder aber nicht genutzte Regeln erkennen.

Zur Analyse Verteilter Anwendungen sind die folgenden vorgestellten Tool nur bedingt geeignet. Diese Tools können nur die Prozesse analysieren, die auf dem selben Knoten laufen, auf dem sie selbst gestartet worden sind

6.1 Testmodul

Zum Testen der Tools ist das Programm der dinierenden Philosophen nicht geeignet, da die Funktionen gleich oft aufgerufen werden. Das folgende Programm berechnet eine gewünschte Fibonacci-Zahl. Die ausführbaren Zeilen sind mit ihren Zeilennummern versehen.

```
-module(fib).
-export([start/1,fib/1]).

start(N) ->
    V = fib(N),                          % 5
    io:format("fib(~w) = ~w~n",[N,V]).  % 6

fib(0) ->
    1;                   % 9
fib(1) ->
    1;                   % 11
fib(N) when N>1 ->
    fib(N-1)+fib(N-2);   % 13
fib(N) ->
    io:format("fib not defined for negative numbers~n",[]). % 15
```

6.2 cprof

Das Modul `cprof` stellt einen Profiler zur Verfügung, der feststellt, wie häufig eine Funktion aufgerufen wurde. Dazu wird jeder Funktion ein Zähler zugewiesen.

Zur Durchführung der Analyse sind folgende Schritte notwendig:

1. `cprof:start()` : Start der Profiler inkl. Nullsetzen der Zähler
2. Ausführung des zu analysierenden Programms
3. `cprof:pause()` : Pausieren des Profilers
4. `cprof:analyse(Modul)` oder `cprof:analyse()` : Analyse für alle Module des Erlang Knoten oder für ein bestimmtes Modul ausgeben
5. `cprof:stop()` : Beenden des Profilers oder
   `cprof:restart()` : Nullsetzen der Zähler, um dann mit Schritt 2 weiterzumachen

Das möglichst schnelle Pausieren des Profilers vor der Analyse ist notwendig, damit im Hintergrund laufende Prozesse das Ergebnis nicht verfälschen. Alle Funktionen des Moduls `cprof` (außer `cprof:analyse`) geben die Anzahl der Funktionen zurück, die mit einem Zähler überwacht werden.

```
Eshell V5.2  (abort with ^G)
1> cprof:start(),fib:start(5),cprof:pause().
fib(5) = 8
3125
2> cprof:analyse(fib).
{fib,16,[{{fib,fib,1},15},{{fib,start,1},1}]}
3> cprof:stop().
3125
```

Die Analyse zeigt, dass die Funktionen des Modul `fib` insgesamt 16 mal aufgerufen worden sind, wobei davon die Funktion zur Berechnung der Fibonacci-Zahl 15 mal aufgerufen ist. cprof zeigt aber nicht an, welche der vier Regeln von `fib` zur Berechnung wie oft benutzt worden ist.

6.3 cover

Das Modul `cover` stellt Funktionen bereit, um festzustellen, welche ausführbare Zeilen in einem Programm wie häufig durchlaufen werden.

Zur Durchführung der Analyse sind folgende Schritte nötig:

1. `cover:start()` : Initialisieren der im Hintergrund laufender Datenbank zur Speicherung der benutzten Programmzeilen
2. `cover:compile_module(Modul)` : Spezielle Compilierung für jedes Modul, das analysiert werden soll
3. Ausführung des zu analysierenden Programms
4. `cover:analyse(Modul,Modus,Detailgrad)` : Analyse für ein Modul nach bestimmten Parametern ausgeben
5. `cover:stop()` : Beenden der Analyse oder
   `cover:reset()` : Reset der Datenbank, um dann mit Schritt 3 wieder zu beginnen

Bei der Analyse eines Moduls wird die Datenbank in einem bestimmten Modus nach einem Detailgrad ausgewertet ausgegeben. Der Detailgrad bestimmt, wie genau die Analyse für ein Modul ausgegeben wird, und lässt sich wie folgt auswählen:

Detailgrad	Ausgabe der Analyse	
`module`	`{Modul, value}`	mit `Modul` = Name des Moduls
`function`	`[{Function,value}]`	mit `Function` = {Modul, Name der Funktion, Stelligkeit}
`clause`	`[{Clause,value}]`	mit `Clause` = {Modul, Funktion, Stelligkeit, Position der Klausel}
`line`	`[{Line,value}]`	mit `Line` = {Modul, Zeilennummer}

Zur Analyse stehen zwei Modi zur Verfügung:

- `coverage` : Es wird festgestellt, ob eine Zeile ausgeführt wurde oder nicht. `value` besteht aus einem Tupel `{Cov,NotCov}`, wobei `Cov` die Anzahl der ausführbaren Zeilen in einem Modul, einer Funktion, einer Regel oder einer Zeile ist, die mindestens einmal ausgeführt worden sind, und `NotCov` die Anzahl der ausführbaren Zeilen, die nicht ausgeführt worden sind.
- `calls` : Es wird festgestellt, wie häufig ein Modul, eine Funktion, eine Regel oder eine ausführbare Zeile aufgerufen wurde. `value` gibt die Anzahl der Aufrufe an.

Das Beispiel bezieht sich auf das oben angegebene Modul `quicksort`.

```
Eshell V5.2  (abort with ^G)
1> cover:start().
{ok,<0.29.0>}
2> cover:compile_module(fib).
{ok,fib}
3> fib:start(5).
fib(5) = 8
ok

4> cover:analyse(fib,coverage,module).
{ok,{fib,{5,1}}}
```

Hier wird deutlich, dass von den sechs ausführbaren Zeilen im Modul `fib` eine nicht benutzt wird.

```
5> cover:analyse(fib,calls,clause).
{ok,[{{fib,start,1,1},1},{{fib,fib,1,1},3}, {{fib,fib,1,2},5},
                         {{fib,fib,1,3},7}, {{fib,fib,1,4},0}]}
6> cover:stop().
```

Im Gegensatz zum Profiler `cprof` kann man durch geeignete Parameter nicht nur erkennen, welche Funktionen wie häufig aufgerufen wurden sondern auch welche Regel einer Funktion wie oft aufgerufen wurde. Die vierte Regel wurde nicht benutzt.

Ein Vorteil der Analyse mit den beiden obigen Profilern ist die Art der Rückgabe des Ergebnisses. Sie erfolgt als Erlang Datenstruktur, so dass man das Ergebnis mit anderen Programmen in Erlang weiterverarbeiten kann. Es entfällt der Vorgang des Parsens, der bei vielen Programmiersprachen notwendig ist, die nicht über geeignete Möglichkeiten der Datendarstellung verfügen.

Die Ausgabe, welche ausführbare Zeile wie häufig aufgerufen wurde (mittels `cover:analyse(fib,calls,clause)`), kann schnell sehr unübersichtlich werden. Deshalb ist auch eine besser formatierte Ausgabe in eine Datei mittels `cover:analyse_to_file(Modul)` möglich. Ein Auszug aus solch einer Datei sieht wie folgt aus:

```
   |  fib(0) ->
3..|      1;
   |  fib(1) ->
5..|      1;
   |  fib(N) when N>1 ->
7..|      fib(N-1)+fib(N-2);
   |  fib(N) ->
0..|      io:format("fib not defined for negative numbers~n",[]).
```

Zuerst wird angegeben, wie oft eine Zeile aufgerufen worden ist, und dann folgt die eigentliche Programmzeile aus dem Quellcode.

6.4 eprof

Das Modul `eprof` stellt Funktionen zur Verfügung, um festzustellen, welche Funktion wie viel Zeit während der Ausführung eines oder mehrere Prozesse benötigt.

Um die Analyse durchführen zu können, sind folgende Schritte notwendig:

1. `eprof:start()` : Start des Eprof Servers und Initialisierung der Datenbank
2. `eprof:start_profiling(Rootset)` : Start der Zeitnahme für die in `Rootset` gespeicherten Prozesse und der Prozesse, die von ihnen erzeugt worden sind
3. Ausführung der zu analysierenden Funktionen
4. `eprof:stop_profiling()` : Ende der Zeitnahme
5. `eprof:analyse()` : Analyse der Datenbank für jeden Prozess einzeln oder `eprof:total_analyse()` : Analyse für alle Prozesse zusammen
6. `eprof:stop()` : Beenden des Servers (oder weiter bei Schritt 2)

Nach der Berechnung der Fibonacci-Zahl von 5 ergibt sich bei `eprof:analyse()` folgender (gekürzter) Bericht:

```
FUNCTION                                          CALLS      TIME

****** Process <0.31.0>    -- 100 % of profiled time ***
fib:fib/1                                         15         38 %
gen:wait_resp_mon/3                               1          6 %
error_handler:undefined_function/3                1          5 %
io:request/2                                      1          4 %
fib:start/1                                       1          4 %
io:format/2                                       1          3 %


Total time: 0.00
```

Im Bericht ist jeder Prozess (in diesem Fall gibt es nur einen) inklusive seiner Pid aufgeführt und wie viel Prozent er von der gemessenen Gesamtzeit beansprucht hat. Für jeden Prozess ist weiterhin dargestellt, welche Funktion wie häufig aufgerufen worden ist und wie lange die Ausführung gedauert hat. Dabei ist die Zeit auch wieder eine Prozentzahl, die angibt, wie viel der gesamten Rechenzeit eines Prozesses von einer Funktion beansprucht wurde.

Die gemessenen Zeiten der Prozessorauslastung sind aber mit Vorsicht zu betrachten, da sich die Laufzeit durch den im Hintergrund laufenden Profiler durchaus verdoppeln kann.

7 Tracer

Unter Tracen versteht man die Überwachung und Protokollierung von Funktionen, Prozessen und Nachrichten. So lässt sich z.B. nachvollziehen, wann ein Prozess mit welchen Parametern neu gestartet worden ist.

Mit den vorgestellten Tracern in den Modulen `dbg` und `ttb` lassen sich sowohl der lokale als auch andere Knoten überwachen. Die anderen Knoten müssen lediglich bei der Initialisierung des jeweiligen Debuggers mit ihrem Namen aufgeführt werden.

7.1 dbg

Im Modul `dbg` wird eine große Anzahl von Funktionen zum Tracen bereit gestellt. Im Gegensatz zum später vorgestellten Modul `ttb` werden hier alle Ausgaben des Traces sofort bei ihrem Auftreten auf der Konsole ausgegeben.

Das Modul `dbg` lässt sich auf viele Weise nutzen. Im folgenden wird eine Möglichkeit des Tracen Schritt für Schritt gezeigt, das eine große Anzahl von Einstellungsmöglichkeiten bietet:

1. `dbg:tracer()` : Ein lokaler Server wird gestartet, der alle Ausgaben des Traces ausgibt.
2. `dbg:n(Node)` : Ein anderer Knoten wird dem Server zum Tracen hinzugefügt. Alle dort anfallenden Nachrichten werden zum Server weitergeleitet und lokal ausgegeben. Der Befehl kann mehrfach oder gar nicht ausgeführt werden.
3. `dbg:tpl(Module,Function,Arity,MatchSpec)` : Eine Funktion aus einem Modul mit einer Stelligkeit wird dem Server zur Überwachung hinzugefügt. Anstatt der Angaben zur Stelligkeit oder den Angaben zur Funktion und zur Stelligkeit können auch Platzhalter (`'_'`) stehen. `tpl` kann auch mehrfach aufgerufen werden, damit verschiedene Module überwacht werden können, oder gar nicht, wenn keine Funktion überwacht werden soll.
   `MatchSpec` bestimmt, was bei einer Funktion getract werden soll. Zum Tracen des Funktionsaufrufes und des Rücksprungs muss
   `[{'_',[],[{message,{caller}},{return_trace}]}]` gewählt werden.
4. `dbg:p(Item, Flags)` : Es wird festgelegt, wonach (`Flags`) wer (`Item`) getract werden soll. Die beiden Parameter lassen sich wie folgt belegen, wobei nicht alle Möglichkeiten angegeben sind:

`Item`	**Wer soll überwacht werden?**
`Pid`	Pid des zu überwachenden Prozesses, wobei der Prozess auf dem lokalen oder auf den mit `dbg:n` spezifizierten Knoten laufen kann
`existing`	alle Prozesse, die auf dem lokalen oder auf den mit `dbg:n` spezifizierten Knoten existieren
`all`	alle Prozesse, die auf dem lokalen oder auf den mit `dbg:n` spezifizierten Knoten existieren und neu gestartet werden
`new`	alle Prozesse, die auf dem lokalen oder auf den mit `dbg:n` spezifizierten Knoten neu gestartet werden

`Flags`	**Wonach soll getract werden?**
`s`	`send` – Tracen nach Nachrichten, die ein überwachter Prozess gesendet hat
`r`	`receive` – Tracen nach Nachrichten, die ein überwachter Prozess geschickt hat
`m`	`message` – Kombination aus `send` und `receive`
`c`	`calls` – Tracen der überwachten Prozesse nach Funktionsaufrufen, die mit `tpl` spezifiert worden sind
`p`	`procs` – Tracen der überwachten Prozesse nach relevanten Events
`clear`	alle vorher gesetzten Flags werden gelöscht

Ab jetzt kann es zu Ausgaben des Tracers kommen.
5. Ausführen des zu tracenden Programms

Die Vorbereitung zum Tracen kann dann wie folgt aussehen:

```
Eshell V5.2  (abort with ^G)
(a@magnus)1> dbg:tracer().
{ok,<0.35.0>}
(a@magnus)2> dbg:n(b@magnus).
{ok,b@magnus}
(a@magnus)3> dbg:n(c@magnus).
{ok,c@magnus}
(a@magnus)4> dbg:p(new,[s]).
{ok,[{matched,b@magnus,0},{matched,c@magnus,0},{matched,a@magnus,0}]}
```

Mit diesen Einstellungen werden die Knoten a@magnus, b@magnus und c@magnus nach gesendeten Nachrichten getract.

```
(a@magnus)5> dining:start(2,a@magnus,[b@magnus,c@magnus]).
```

Nach dem obigen Aufruf endet der stark gekürzte Trace folgendermaßen:

```
(a@magnus)6> 0 took left stick
(a@magnus)6> 1 took left stick
(a@magnus)6> (<3844.121.0>) <0.48.0> ! {take,<3844.121.0>}
(a@magnus)6> (<3926.117.0>) <0.49.0> ! {take,<3926.117.0>}
```

Die Ausgabe des Traces zeigt, dass die beiden Prozesse der Philosophen ihren jeweiligen linken Stick genommen haben und eine Nachricht geschickt haben, dass sie ihren rechten Stick haben wollen. Der Deadlock ist eingetreten.

7.2 Trace Tool Builder

Der Trace Tool Builder ttb ist eine Weiterentwicklung des Tracers im Modul `dbg`. Der große Unterschied ist, dass die Ausgaben des Tracers in einer Datei zwischengespeichert werden und nach Ablauf des Programms formatiert und ausgegeben werden können. ttb bietet viele Einstellungsmöglichkeiten, wobei hier nur einige aufgezählt sind.

Die Anwendung von ttb ähnelt sehr der Anwendung von dbg:
1. `ttb:tracer(Nodes)` : Auf jedem der angegeben Knoten wird ein Server gestartet, der wiederum eine Datei mit dem Namen des Knotens plus `-ttb` zum Speichern des Traces anlegt.
2. `ttb:tpl(Module,Function,Arity,MatchSpec)` : Das Hinzufügen von Funktionen erfolgt mit den gleichen Einstellungsmöglichkeiten genau wie bei `dbg:tpl`.
3. `ttb:p(Item, Flags)` : Auch das Festlegen der Art des Tracens erfolgt wie bei `dbg:p`.
4. Ausführen des zu testenden Programms
5. `ttb:stop()` : auf allen Knoten die Tracer anhalten

Nun existiert auf jedem getracten Knoten eine Datei im binären Format, die ausgewertet werden kann. Die einfachste Art der Formatierung und Ausgabe ist: ttb:format(Dateiname).
Die Ausgabe erfolgt dann sehr ähnlich wie beim Tracer des Moduls `dbg`. Allerdings wird nur ein Knoten betrachtet. Wenn man mehrere Knoten auf einmal analysieren will, muss statt dem Dateinamen eine Liste von Dateinamen angegeben werden.

Statt der Standardausgabe kann man aber auch eigene Ausgabefunktionen definieren, weshalb dieser Tracer Trace Tool Builder heißt. Der Aufruf lautet dann:

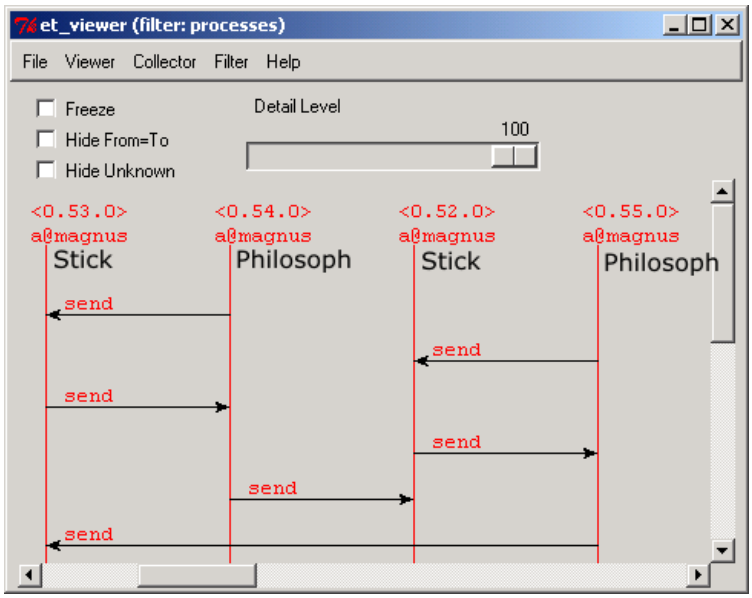
`ttb:format(Dateiname,[{handler,FormatHandler}])`, wobei dann jede Zeile des Traces mit der im `FormatHandler` angegebenen Funktion ausgegeben wird. Ein vorhandener `FormatHandler` ist der später vorgestellte Event Tracer.

Beim Vergleich der Tracer ttb und dbg wird deutlich, dass ttb wesentlich bessere Möglichkeiten zur Ausgabe des Traces hat, während dbg keine großen Eingriffe in die Performance des Systems macht, da keine Benutzung von Festplatten erfolgt.

7.3 et

Der Event Tracer et tract keine Anwendungen, sondern gibt Ereignisse (auch Trace-Nachrichten) graphisch wieder.

Ein mittels ttb erzeugter Trace für gesendete Nachrichten vom Aufruf von `dining:start(2,a@magnus,[a@magnus,a@magnus])` erzeugt nach Aufruf des Event Trackers mit `ttb:format("a@magnus-ttb",[{handler,et}])` eine Message Sequence Chart (MSC):



Man sieht, dass die beiden Philosophen nach dem linken Stick greifen wollen und ihn auch bekommen. Dann greifen sie nach dem rechten Stick, den sie aber nicht bekommen. Es ist ein Deadlock aufgetreten.

Unabhängig vom Beispiel werden Ereignisse an den gerichteten Pfeilen und der roten Schrift erkennbar. Ereignisse, die denselben Start- und Endpunkt haben, werden mit blauer Schrift und ohne Pfeil dargestellt. Die Reihenfolge der Ereignisse wird durch ihre Position im Diagramm wiedergegeben.

Das Diagramm lässt sich auf folgende Weise beeinflussen:

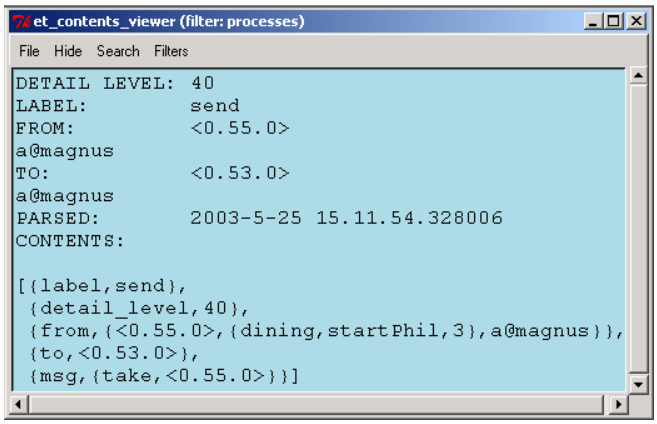
- Detail Level: Ein Ereignis wird nicht dargestellt, wenn sein `DetailLevel` größer als der im Schieberegler eingestellte Wert ist.
- Freeze: neue Ereignisse werden nicht mehr angezeigt
- Hide From=To: Ereignisse mit gleicher Start- und Endpunkt werden nicht angezeigt.
- Hide Unknown: Ereignisse mit unbekanntem Start- oder Endpunkt werden ausgeblendet

Mittels Drag & Drop lassen sich Ereignispunkte in der Graphik verschieben und beim Anklicken eines oder mehrere Ereignispunkte werden nur noch deren Ereignisse angezeigt.

Im Menü Viewer kann man, wenn nicht alle Ereignisse aus Platzgründen auf einmal dargestellt werden können, auf diverse Arten durch die Liste der Ereignisse scrollen.

In diesem Beispiel werden alle Ereignisse in einer Graphik dargestellt. Da aber beim Tracen eines Programms nach verschiedenen Gesichtspunkten getract werden kann, lassen sie sich mittels Filter im Menü Filter von einander trennen und in verschiedenen Fenstern darstellen. Um die

Ereignisse in verschiedenen Fenstern zu gleichen Zeitpunkten sehen zu können, kann man die Darstellung der Ereignisse in allen Fenstern gleichzeitig im Menü Collector beeinflussen.



Alle bekannten Informationen zu einem Ereignis lassen sich durch Anklicken eines Ereignisses anzeigen.

Unter anderem wird im Beispiel auch der Text der Nachricht angezeigt.

Im Menü Hide lässt sich die Darstellung der Ereignispunkte in der Graphik beeinflussen.

Der Event Tracer lässt sich auch zur Darstellung selbst generierter Ereignisse nutzen. Zuerst wird die graphische Anzeige mittels `et_viewer:start([])` gestartet. Bevor die Ereignisse im Event Tracer dargestellt werden können, muss ein Prozess gestartet werden, der sie speichert: `et_viewer:get_collector_pid(ViewerPid)`. `ViewerPid` ist die Pid der graphischen Anzeige und der Rückgabewert der Funktion ist der Pid des Prozesses, der die Ereignisse sammelt.

Zum Speichern der Ereignisse kann folgende Funktion aufgerufen werden:
`et_collector:report_event(CollectorPid,DetailLevel,From,To,Label,Contents)`

Die Parameter bedeuten folgendes:
- `CollectorPid` : Pid des Prozesses, der die Ereignisse sammelt
- `DetailLevel` : Zahl zwischen 0 und 100
- `From` : Startpunkt des Ereignisses
- `To` : Zielpunkt des Ereignisses (kann auch weggelassen werden)
- `Label` : Name des Ereignisses
- `Contents` : Beschreibung des Ereignisses

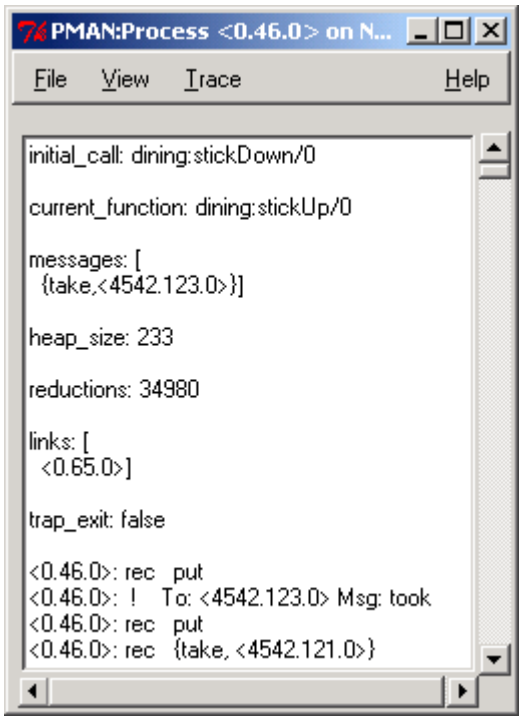
Sobald die Funktion `report_event` aufgerufen worden ist, wird die Anzeige aktualisiert.

7.4 Tracen mit dem Process Manager

Auch der Process Manager lässt sich zum Tracen nutzen. Alle im Process Manager angezeigten Prozesse lassen sich mittels eines Doppelklicks auf den gewünschten Prozess tracen. Es wird ein Fenster geöffnet, in dem der interne Tracer Informationen ausgibt.

Im Trace Fenster werden zusätzlich zum Process Manager folgende Informationen angezeigt werden, die laufend aktualisiert werden, falls Veränderungen aufgetreten sind:

- initial call: die Funktion (Modul:Funktion/Stelligkeit), die der Prozess zum Starten aufgerufen hat
- messages: Nachrichten in der Mailbox, die noch nicht bearbeitet worden sind
- heap size: Größe des Heaps
- stack size: Größe des Stacks
- links: List der Pids mit denen der Prozess verlinkt ist
- trap_exit: Reaktion auf ein Exit Signal
  `true` – Signal wird in Nachricht umgewandelt und Prozess terminiert nicht
  `false` – Prozess terminiert
- Trace-Ausgaben



Eine Trace-Ausgabe beginnt mit der Pid des überwachten Prozesses. Dann folgt ein unten näher beschriebenes Tag mit zusätzlichen Informationen:

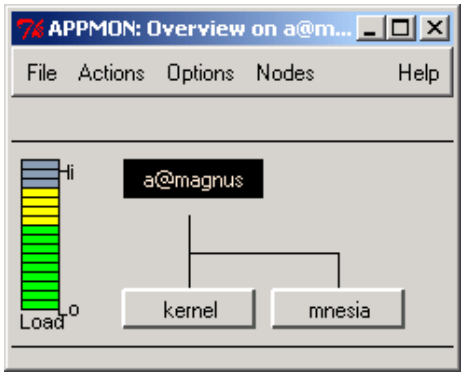
Tag	**Erläuterung**	**zusätzliche Informationen**
`!`	Eine Nachricht wurde gesendet.	Pid des Empfängers und gesendete Nachricht
`rec`	Ein Prozess hat eine Nachricht aus der Mailbox genommen.	empfangene Nachricht
`call`	Aufruf einer Funktion	Name der Funktion mit allen Parameter
`link`	Ein Link wurde zwischen dem überwachten und einem anderen Prozess erstellt.	Pid des anderen Prozesses
`spawn`	Ein neuer Prozess wurde vom überwachten Prozess gestartet.	Pid des gestarteten Prozesses
`exit`	Der überwachte Prozess wurde terminiert.	Grund der Terminierung

Im Menü View lassen sich Informationen zum Modul anzeigen. Die Trace Fenster von verlinkten Prozessen lassen sich vom Menü Trace aus aufrufen. Des weiteren kann man den Prozess beenden.

8 Application Monitor

Der Application Monitor appmon gibt eine graphische Übersicht über alle auf einem Knoten laufenden Applikationen aus. Eine Applikation ist eine Anwendung, die nach den Erlang/OTP Design Prinzipien programmiert worden ist. Eine Erläuterung der Design Prinzipien würde den Rahmen dieser Ausarbeitung sprengen, sie sind aber in der Online Dokumentation beschrieben.

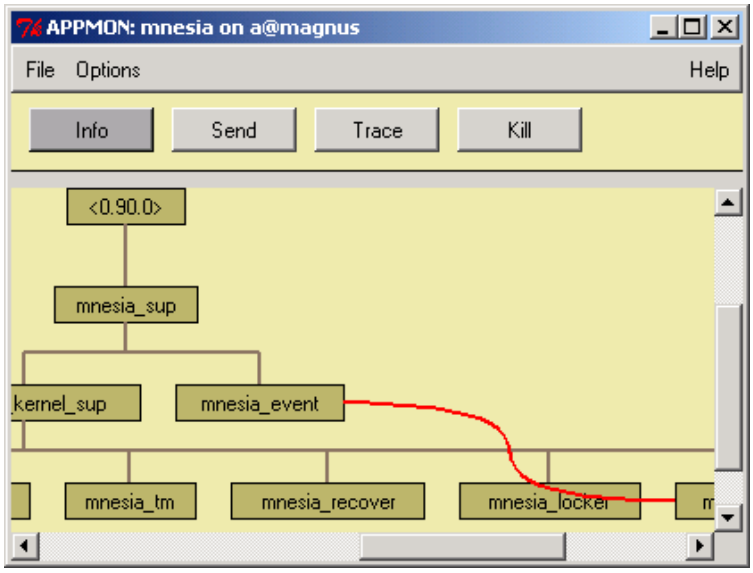
Ein Aufruf von `appmon:start()`, nachdem die Applikation Mnesia (eine Datenbank) gestartet worden ist, ergibt folgende Anzeige:



Am linken Rand wird die Systemauslastung angezeigt. In der Mitte steht der aktuelle Knoten, der überwacht wird, und unter ihm sind die laufenden Applikationen als Buttons angezeigt.

Im Menü Actions lässt sich der aktuelle Knoten neu starten, beenden oder anpingen, und im Menü Options kann man die Anzeige verändern. Im Menü Nodes kann ein anderer Knoten zu Überwachung ausgewählt werden. Es werden aber nur Knoten angezeigt, die dem aktuellen Knoten auf irgendeine Weise bekannt gemacht worden sind.

Wenn man einen Button drückt, der eine Applikation darstellt, wird ein neues Fenster mit einer hierarchischen Ansicht der Prozesse der gewählten Applikation angezeigt. Dieser Baum wird Supervision Tree genant.



Jeder Prozess (abgesehen vom ersten) hat genau einen Supervisor Prozess. Aber ein Supervisor Prozess kann mehrere Prozesse unter sich haben.

Diese Abhängigkeit wird in Erlang durch das Verlinken von Prozessen erzeugt, und in dieser Graphik wird der Link durch eine braune Line angezeigt. Verlinkte Prozesse, die in keiner Abhängigkeit im Supervision Tree zueinander stehen, werden mit einer roten Linie verbunden.

Die Buttons oberhalb des Supervision Trees benutzt man, indem zuerst ein Button ausgewählt und dann ein Prozess angeklickt wird. Sie haben folgende Funktion:

- Info: Anzeigen von Informationen über den Prozess, die von `erlang:process_info` generiert werden
- Send: Dem Prozess wird eine Nachricht gesendet. Über ein Dialogfenster kann die Nachricht näher spezifiziert werden.
- Trace: Der Prozess wird mittels eines integrierten Tracer getract. Die Ausgabe des Traces erfolgt auf der Shell.
- Kill: Der Prozess wird terminiert.

9 Zusammenfassung

In dieser Ausarbeitung wurden Tools vorgestellt, die Prozesse überwachen, debuggen, profilen und tracen können. Diese Tools lassen sich ohne Einschränkungen zur Analyse von nebenläufigen Systemen einsetzen.

Bei der Analyse von Verteilten Systemen lassen sich die Tools, die Prozesse und Applikationen überwachen und tracen, sehr gut einsetzen. Es muss lediglich der Knoten angegeben werden, der

kontrolliert werden soll. Der Debugger hingegen kann auf einem anderen Knoten nur Prozesse überwachen, die ein lokaler Prozess dort gestartet hat.

Ein wünschenswertes Tool wäre eine Kombination aus dem Application Monitor, einem Tracer und dem Debugger. In einer Graphik sollten alle Knoten zu sehen sein, die dem lokalen Knoten bekannt sind. Nach dem Markieren eines Knotens müssten sich die dort laufenden Prozesse inklusive ihrer Links zu anderen Prozessen (auch auf anderen Knoten) einblenden lassen. Das setzt natürlich gut einstellbare Ausgabefilter voraus, damit man vor lauter Prozessen die Übersicht nicht verliert. Nach dem Anklicken eines Prozesses sollte man in einen Debug-Modus gelangen, so dass man in einem extra Fenster den Prozess auf Wunsch schrittweise kontrollieren kann. Im Hauptfenster wird dabei natürlich die Kommunikation mit anderen Prozessen graphisch angezeigt. Alternativ zum Debug-Modus sollte man den Trace diverser Prozesse in eine Datei speichern können, damit man die Aktionen der Prozesse nachher in einem Wiedergabe-Modus in der Graphik kontrollieren kann.

Auch wenn nicht alle relevanten Tools in der OTP Library die Entwicklung von Verteilten Systemen unterstützen, können die Tools zur Erstellung von Anwendungen in Verteilten Systemen hilfreich sein.