Sprachen für die Parallelprogrammierung: Erlang

Nils Adermann

Abstract
Erlang ist eine funktionale Programmiersprache, die Aktoren zur Umsetzung von Nebenläufigkeit und Fehlertoleranz einsetzt. Aufgrund der Herkunft aus dem Telekommunikationsumfeld bietet die Sprache zahlreiche Mechanismen zur Entwicklung hoch skalierbarer und fehlertoleranter verteilter Systeme. Mit dem Aufstieg des Internets hat Erlang ein neues Anwendungsgebiet gefunden, in dem redundante Systeme große Mengen von Verbindungen parallel bedienen müssen.

1 Erlangs Entwicklungsgeschichte

1.1 Zielsetzung

Die Entwicklung Erlangs beginnt 1986 im Ericsson Computer Science Laboratory mit der Aufgabe die Entwicklung von Telefonieanwendungen zu verbessern. Im selben Labor war bereits der AXE Telefonvermittler mit der Programmiersprache PLEX entwickelt worden. PLEX ist eine auf die AXE Platform zugeschnittene Programmiersprache, die zwar viele Probleme aus AXE Vorgängern behebt, aber nicht für Anwendungen einsetzbar ist, die auf anderer Hardware betrieben werden sollen. Erlang soll deshalb insbesondere auch auf gewöhnlicher Hardware verwendet werden können [1].

Die zentralen Forderungen an Erlangs Spracheigenschaften ergeben sich dann aus der Aufgabe, für die Erlang eingesetzt werden soll. Telekommunikationssysteme müssen dazu in der Lage sein, eine Vielzahl von Verbindungen gleichzeitig aufrecht zu erhalten. Sie bestehen oft aus verteilten Systemen um die anfallenden Datenmengen verarbeiten zu können und um Redundanz für Fehlerfälle bereit zu stellen. Die Sprache muss also auf verteilten Systemen einsetzbar sein. Sie muss außerdem eine einfache Interaktion mit der Hardware erlauben, da insbesondere Telekommunikationsdienste, die große Datenmengen verarbeiten müssen, aus Geschwindigkeitsüberlegungen auf spezialisierte Hardware-Komponenten setzen. In solchen Systemen treten auch zeitkritische Aktivitäten auf, Erlang muss also weiche Echtzeitbedingungen erfüllen können.

Die Anforderung an die kontinuierliche Laufzeit eines Telekommunikationssystems kann mehrere Jahre umfassen. Es müssen also Mechanismen für die Wartung der Software bereit gestellt werden, die eine Unterbrechung der Funktionalität vermeiden.

Erlang muss in mehrerer Hinsicht fehler-

tolerant sein: Beim Auftreten eines Softwarefehlers in der Verarbeitung einer Verbindung müssen die restlichen laufenden Verbindungen fehlerfrei fortgeführt werden. Beim Auftreten eines Hardwarefehlers im verteilten System müssen Aufgaben dynamisch auf andere Hardware umverteilbar sein.

Die Sprache soll außerdem die Entwicklung komplexer Systeme, wie sie in der Telekommunikationsbranche angetroffen werden, mit mehreren Millionen Zeilen Programmtext in größeren Teams unterstützen.

1.2 Erste Versuche mit anderen Sprachen

Zunächst wurden Experimente mit diversen bereits existierenden Programmiersprachen durchgeführt. Zu diesem Zeitpunkt gab es eigentlich keine Absicht eine neue eigene Sprache zu entwickeln. Im Rahmen dieses Projekts gelangte die Gruppe zur Feststellung, dass deklarative regelbasierte Logiksprachen wie die untersuchten Sprachen PFL und LPL0 die elegantesten Programme hervorbringen. Sie verstanden Nebenläufigkeit als essentiell für Telekommunikationsanwendungen, mussten aber zur Kenntnis nehmen, dass Nebenläufigkeit in deklarativen Sprachen bislang wenig erforscht wurde.[1]

Joe Armstrong beschäftigte sich im Rahmen dieser Experimente mit Smalltalk und Prolog. Das zu lösende Problem hatte er graphisch notiert und zunächst in Smalltalk implementiert, wobei er Probleme mit der Geschwindigkeit Smalltalks und der Garbage-Collection der Laufzeitumgebung bemerkte. Ein Kollege wies ihn auf die Ähnlichkeit seiner graphischen Notation zu einem Prolog Programm hin und er konnte es tatsächlich in lediglich 15 Zeilen umsetzen. Beeindruckt begann er sich mehr mit Prolog auseinander zu setzen.[1]

1.3 Von Prolog zur eigenständigen Sprache

Prologs größtes Hindernis für den Einsatz in Telekommunikationssystemen stellt das fehlen jeglichen Konzeptes der Nebenläufigkeit dar. Armstrongs erster Ansatz für die Einführung von Nebenläufigkeit in Prolog war mittels eines Prolog Meta-Interpreters ein Prozess-Konzept umzusetzen, in dem mehrere Ausführungsstränge jeweils über eine Liste von zu erfüllenden Zielen verfügen, die unabhängig voneinander verarbeitet werden können. Im nächsten Schritt führte er eine einfache Form der Kommunikation über Nachrichten zwischen den Prozessen ein. Der Interpreter wurde so zunehmend komplizierter. Mit Kollegen wurde der Interpreter der Sprache, die nun Erlang hieß, schrittweise weiter- und zunehmend von ihrem Ausgangspunkt Prolog wegentwickelt. So unterschied sich vor allem Erlangs Umgang mit Fehlern grundlegend von dem in Prolog [2].

In den folgenden Jahren wurde die Sprache nach Rückmeldungen aus Test- und später tatsächlicher Produktentwicklung iterativ auf die Bedürfnisse der Enwickler verschiedener Gruppen innerhalb Ericssons angepasst. Nach 1988 wurde die Sprache selbst allerdings nur noch geringfügig verändert und das Hauptaugenmerk der Sprachentwicklung verlagerte sich auf Kompilierung, Laufzeitumgebung und Werkzeuge zur Unterstützung der Erlang einsetzenden Entwickler. Der Prolog Interpreter wurde schnell durch eine speziell für Erlang entwickelte virtuelle Maschine namens JAM und einen zugehörigen Byte-Code Compiler ersetzt [2].

1.4 Open Source Erlang

1998 wurden ein GPRS System und der Ericsson AXD301 Switch vorgestellt, die hauptsächlich in Erlang programmiert wurden. Der Erlang Quelltext dieser Projekte umfasst trotz der prägnanten Notation Erlangs mehrere Millionen Zeilen Programmtext. In diesem Rahmen konnte empirisch gemessen werden, dass Erlang die Produktivität um ein 4-faches gesteigert hat. Die gemessene Verfügbarkeit des AXD301 liegt bei „nine-nines" [1]. Die Ziele der Entwicklung Erlangs sind somit erreicht worden.

Im selben Jahr beschloss Ericsson aber auch, dass die Entwicklung und Wartung einer eigenen Programmiersprache, zugehöriger Laufzeitumgebung und Entwicklungswerkzeugen nicht ausreichend rentabel sei. Daraufhin durfte Erlang in neuen Projekten nicht mehr eingesetzt werden. Ericsson war aber dazu bereit die Sprache nun unter einer Open Source Lizenz öffentlich verfügbar zu machen.

Während des IT Booms um das Jahr 2000 wurde Erlang von einigen kleineren Firmen weiter benutzt. Die Firma Bluetail AB übernahm die Entwicklung des Erlang Systems und viele der bisher für Ericsson arbeitenden Entwickler. Die zunehmende Bedeutung des Internets schaffte Erlang in den darauf folgenden Jahren einen neuen Markt. Internetsysteme wie Mail- und Webservices verlangen ähnlich wie Telekommunikationssysteme nach hoher Verfügbarkeit und Skalierbarkeit. So entstanden eine Reihe von Lösungen für verschiedene Internetdienstleistungen, die vor allem bei den größten Webdienstleistern mit besonders großen Anforderungen zum Einsatz kommen [1]. Die Performanz Erlangs spielt in diesem Zusammenhang eine immer größere Rolle, sodass heute mit HiPE auch ein Compiler für nativen Maschinencode in Erlang/OTP verfügbar ist [3].

2 Erlang: Sequentiell und Funktional

Erlang ist eine funktionale, also deklarative, Programmiersprache. In Erlang beschreibt man deshalb im Gegensatz zu imperativen Programmiersprachen umgangssprachlich eher *was* passiert, als *wie* es umzusetzen ist. Zuweisungen sind referenziell transparent: Eine Zuweisung der Form `X = X + 1` ist also fehlerhaft. Erlang-Programme werden in Module unterteilt, die Funktionen für die Verwendung in anderen Modulen exportieren.

2.1 Funktionen: Pattern-Matching und Guards

```
-module(math)
-export([fac/1])

fac(N) when N > 0
    -> N * fac(N - 1);
fac(0)
    -> 1.
```

Programmausschnitt 1: Die Fakultätsfunktion in Erlang

Das Fakultätsprogramm in Programmausschnitt 1 exportiert eine Funktion mit dem Namen *fac* und einem Argument (Arität 1), also `fac/1`. Die Funktion wird durch zwei partielle Definitionen definiert. Bei einem Aufruf der Funktion wird in der angegebenen Reihenfolge versucht eine der partiellen Definitionen zu verwenden. Dabei wird *Pattern-Matching* auf die Argumente angewandt, so kann die zweite partielle Funktion im Beispiel nur ausgeführt werden, wenn das Argument 0 ist. Variablennamen beginnen in Erlang immer mit einem Großbuchstaben. Die erste partielle Definition kann

also zunächst für alle Argumente verwendet werden, der sogenannte *Guard* `when N > 0` spezifiziert allerdings eine weitere Bedingung die nicht durch *Pattern-Matching* ausdrückbar ist, sodass diese partielle Definition nicht für Zahlen kleiner oder gleich Null verwendet werden kann.

2.2 Atome und Tupel

Atome in Erlang repräsentieren nichtnumerische Konstanten. Sie beginnen mit einem Kleinbuchstaben, sind immer global und haben keinen zugeordneten numerischen Wert, der an ihrere Stelle verwendet werden kann. Im Gegensatz zu Variablen gibt es aber derzeit in Erlang keine Garbage Collection für Atome. Armstrong beschreibt die Erweiterung der Garbage Collection auf Atome aber in [1] als wünschenswert.

Tupel haben eine fixe Anzahl von Werten beliebigen Typs. Oft wird wie in Programmausschnitt 2 ein Atom zur Identifikation einer Tupel-Struktur verwendet, da Erlangs dynamisches Typsystem keine benutzerdefinierten Typen kennt.

```
Point = {point, 2, 3}.

% Dieser Ausdruck ist fehlerhaft:
{person, FirstName, Name} = Point.

% Dieser Ausdruck extrahiert
% 2 und 3 nach X und Y
{point, X, Y} = Point.
```

Programmausschnitt 2: Verwendung von Tupeln in Erlang

2.3 Listen

Eine Liste kann Daten beliebiger verschiedener Typen beinhalten. Auch für Listen gilt die referenzielle Transparenz, das heißt alle Operationen, die eine Liste verändern, erstellen tatsächlich eine Kopie der Liste. Allerdings können Operationen wie das Anhängen an eine Liste vom Compiler optimiert werden, wenn die alte Liste danach nicht mehr verwendet wird.

Listen können mit dem Pattern `[Head1, Head2, ..., HeadN | Tail]` in eine Anfangsfolge und eine Restliste zerlegt werden. In den meisten Fällen, in denen man in einer imperativen Sprache Iteratoren einsetzen würde, verwendet man in Erlang eine Funktion, die mit diesem Pattern das erste Element der Liste entnimmt um es zu verarbeiten und dann rekursiv die selbe Funktion auf den Rest der Liste anwendet.

```
qsort([]) -> [];
qsort([Pivot|Tail]) ->
    qsort([X ||
        X <- Tail, X < Pivot])
    ++ Pivot ++
    qsort([X ||
        X <- Tail, X >= Pivot]).
```

Programmausschnitt 3: Quicksort mit *List Comprehensions*

Listen können auch mittels sogenannter *List Comprehensions* erzeugt werden. Diese entnehmen zunächst Elemente aus einer oder mehreren Ausgangslisten, überprüfen dann eine beliebige Menge von Bedingungen und fügen sie dann optional transformiert in eine neue Liste ein. Im Programmausschnitt 3 wird die Verwendung von *List Comprehensions* in einer Quicksort Implementierung demonstriert.

Listen werden in Erlang auch für die Repräsentation von Zeichenketten verwendet. Bei großen Datenmengen empfiehlt sich stattdessen die Verwendung von Binaries (siehe nächster Abschnitt), die deutlich speichereffizienter sind.

2.4 Bit Syntax

Die Bit Syntax ist eine spezielle Syntax für das *Pattern-Matching* von Daten des

Typs Binary. Dieses etwas ungewöhnliche Sprachkonstrukt lässt sich leicht durch die Herkunft aus dem Telekommunikationsumfeld erklären, in dem die Verarbeitung von Paketdaten eine wichtige Rolle spielt. Programmausschnitt 4 zeigt den, im Vergleich zu anderen Sprachen kurzen, für das Entpacken eines IPv4 Datagrams notwendigen Programmtext.

```
DgramSize = size(Dgram)
case Dgram of
  <<?IP_VERSION:4, HLen:4,
    SrvcType:8, TotLen:16,
    ID:16, Flgs:3, FragOff:13,
    TTL:8, Proto:8, HdrChkSum:16,
    SrcIP:32,
    DestIP:32, RestDgram/binary>>
      when HLen >= 5,
           4*HLen =< DgramSize
        -> OptsLen =
          4*(HLen - ?IP_MIN_HDR_LEN),
        <<Opts:OptsLen/binary,
          Data/binary>> =
            RestDgram,
        ...
```

Programmausschnitt 4: Entpacken eines IPv4 Datagrams mit Bit Syntax

2.5 Bedingungskonstrukte

Die einzige bislang vorgestellte Möglichkeit Code bedingt auszuführen ist die Auswahl einer partiellen Funktion durch *Pattern-Matching*. Die Verwendung von *Pattern-Matching* ist allerdings auch innerhalb einer Funktion möglich, sodass man nicht für jede Bedingung neue Funktionen definieren muss. Die dafür vorgesehenen Konstrukte heißen `case` und `if`. `case` erlaubt *Pattern-Matching* und *Guards* während `if` nur *Guards* verwendet, dabei kann das Atom `true` wie ein *else* in C verwandten Sprachen verwendet werden. Programmausschnitt 5 demonstriert beispielhaft die Verwendung von `if` innerhalb einer Funktion.

```
remove_neg([H|T]) ->
    if H >= 0 ->
           [H|remove_neg(T)];
       true ->
           [remove_neg(T)]
    end;
remove_neg([]) -> [].
```

Programmausschnitt 5: Eine Funktion, die mit Hilfe von `if` negative Zahlen aus einer Liste entfernt

2.6 Fehlerbehandlung

Erlang verwendet Exceptions, die auf drei verschiedene Weisen erzeugt werden können. Wenn ein Fehler auftritt, der vom Benutzer einer Funktion erwartet wird, sollte `throw/1` verwendet werden. Die Exception kann dann mittels `try ... catch` behandelt werden. Wenn ein unerwarteter Fehler auftritt und der Entwickler den aktuellen Prozess deshalb terminieren möchte, sollte `exit/1` angewandt werden. Erlang selbst wirft interne Fehler mit `erlang:error/1`, diese werden normalerweise nicht erwartet und deshalb auch nicht aufgefangen [4].

```
try FuncOrExpressionSeq of
    PatternN [when GuardN]
        -> ExpressionsN;
    ...
catch
    ExceptionTypeN: ExPatternN
        [when ExGuardN]
            -> ExExpressionsN;
    ...
after
    AfterExpressions
```

Programmausschnitt 6: Allgemeine Form des `try ... catch` Konstrukts

Wie Programmausschnitt 6 zu entnehmen ist, ähnelt die Syntax des `try ... catch` Konstrukts stark dem `case` Ausdruck. Der `ExceptionType` kann `throw`, `exit` oder `error` sein; wird er ausgelassen wird `throw` angenommen. Sollte der Ausdruck, der im `try` Block ausgewertet

wird, nicht weiter verarbeitet werden müssen, kann man die Patterns im `try` Block auch auslassen. Solange keine Exception auftritt, entspricht das Resultat des `try ... catch` Konstrukts dann dem Resultat der Ausführung von `FuncOrExpressionSeq`. Anweisungen im `after` Block werden unabhängig vom Erfolg oder Misserfolg des `try` Ausdrucks immer ausgeführt. Sie sind allerdings ebenfalls optional.

3 Nebenläufigkeitsorientierte Programmierung

3.1 Das Aktormodell

Der einzige Mechanismus zur Realisierung von Nebenläufigkeit in Erlang sind Aktorsysteme. Sie bestehen aus einer Menge von Aktoren, die miteinander kommunizieren und ihr Verhalten abhängig von erhaltenen Nachrichten verändern können. Jeder Aktor besitzt eine Mailadresse, die ihn eindeutig identifiziert und mittels der man mit ihm kommunizieren kann. Ein Aktor reagiert auf ihm gesendete Nachrichten indem er sequentiellen Code ausführt. Darin darf er mehrere Nachrichten an andere Aktoren, deren Adressen er kennt, oder an sich selbst versenden und kann neue Aktoren erstellen. Im Anschluss gibt der Aktor ein Verhalten an, mit dem die nächste eintreffende Nachricht zu verarbeiten ist. [5]

Aktorsysteme abstrahieren das Konzept der Nebenläufigkeit stark und stellen keine Synchronisationsprimitiven zur Verfügung. Die Kommunikation mittels Nachrichten ist in der physikalischen Welt alltäglich. Die Übermittlung einer Nachricht kann nur als gesichert gelten, wenn der Empfänger ihren Empfang durch eine weitere Nachricht bestätigt. Dieses natürliche Konzept wird als deutlich besser mit einer zunehmend verteilten Informationswelt vereinbar angesehen als traditionelle Mechanismen zur Implementierung von Nebenläufigkeit und gilt deshalb als zukunftsweisend [6].

Da jegliche Kommunikation mittels *Message-Passing* durchgeführt wird, teilen die Aktoren keinen gemeinsamen Zustand. Aktoren erhalten Kopien der Nachrichten und keine Referenzen oder Zeiger auf modifizierbare Speicherbereiche, auf die andere Aktoren zugreifen könnten. Man spricht auch von einer *Shared-Nothing* Architektur. Ein Aktorsystem kann deshalb problemlos ohne Anpassungen sowohl auf einem Einprozessorsystem als auch auf verteilten Systemen mit geteiltem oder getrenntem Speicher auf die gleiche Art verwendet werden. Die Verteilung der Aktoren ist also transparent und der Wechsel auf ein verteiltes System bereitet dem Entwickler keinen Mehraufwand. Grundlage dafür ist allerdings die konsequente Verwendung von Aktoren in allen Bereichen der Anwendung.

Die Benutzung von Aktoren auch für Teile des Programms, die nicht notwendigerweise nebenläufig ausgeführt werden müssen liegt im Interesse des Entwicklers, da Aktoren insbesondere auch der Trennung verschiedener Systemkomponenten zur Steigerung der Fehlertoleranz dienen. Tritt ein Fehler in einem Aktor auf kann der Rest des Systems weiter arbeiten während der fehlerhafte Aktor neu gestartet wird.

Für den konsequenten Einsatz von Aktoren für alle Teilkomponenten eines Systems prägten Armstrong et. al. den Begriff *Concurrency Oriented Programming*, zu deutsch *nebenläufigkeitsorientierte Programmierung*, in Anlehnung an die objektorientierte Programmierung. Es gibt wie in [5] beschrieben auch eine konzeptionelle Ähnlichkeit der objektorientierten Pro-

grammierung zur nebenläufigkeitsorientierten Programmierung: Aktoren kapseln internen Zustand, der mittels Nachrichten oder Methodenaufrufen transformiert werden kann. Aktorsysteme können auch Polymorphie und andere objektorientierte Konzepte umsetzen, dies ist in Erlang allerdings nicht der Fall. So ist Erlang zwar prinzipiell keine objektorientierte Programmiersprache, bietet aber mit Aktoren ein eingeschränkt ähnliches Konzept.

Die Implementierung von Aktorsystemen in Erlang wird als „de facto" Implementierung des Aktormodells angesehen und dient neueren Implementierungen als Referenzpunkt bezüglich Funktionalität, Performanz und Skalierbarkeit [7]. Die virtuelle Maschine Erlangs hat spezielle Instruktionen für die Umsetzung von Aktorsystemen [8]. Aktuelle Forschung zur Implementierung von Aktorsystemen auf der Java Virtual Machine, die keine solchen Instruktionen bietet (Scala Actors, Kilim, ActorFoundry, etc.) reicht deshalb noch nicht an die Skalierbarkeit Erlangs heran [9].

3.2 Prozesse: Aktoren in Erlang

Aktoren in Erlang heißen Prozesse und ihre Mailadressen werden als Prozess-Identifikatoren bezeichnet. Diese Prozesse werden nicht auf Systemprozesse oder -threads abgebildet, sondern lediglich innerhalb der virtuellen Maschine getrennt verarbeitet. Ein Prozess benötigt nur wenige Byte Speicher und kann deshalb im Gegensatz zu Systemprozessen oder -threads sehr schnell erstellt werden. Deshalb ist es insbesondere kein Problem mehrere Tausend Erlang Prozesse zu erstellen wodurch Entwicklern die Entscheidung für Aktorsystemen zur Modularisierung eines Programms erleichtert wird. In neueren Versionen Erlangs wird ein Hybridsystem eingesetzt, das mehrere Systemthreads einsetzt, aber keine 1-zu-1 Abbildung von Prozessen durchführt um die Leichtigkeit von Prozessen bei zu behalten, vorher betrieb man auf Mehrkernprozessoren mehrere Erlang Instanzen (*Nodes*).

Die Erlang Funktion `spawn/1` erstellt einen neuen Prozess, der die als Parameter übergebene Funktion ausführt. Soll der neue Prozess initialisiert werden so kann man wie in Programmausschnitt 7 eine anonyme Funktion mit `fun` erstellen, die lediglich die eigentliche Prozessfunktion mit den Initalisierungsparametern aufruft. Das Resultat des Funktionsaufrufs ist ein Prozess-Identifikator, der später zur Kommunikation mit dem Prozess verwendet werden kann. Eine Nachricht `Message` wird mittels `Pid ! Message` an den Prozess mit dem Identifikator `Pid` gesendet.

```
ask() ->
    Pid = spawn(fun()
        -> server(InitialState)),
    Pid ! {self(), Question}.
```

Programmausschnitt 7: Erstellt Instanz des Servers aus Programmausschnitt 8 und stellt eine Anfrage

Der Empfang von Nachrichten wird durch das `receive` Konstrukt abgewickelt. Mittels *Pattern-Matching* wird bestimmt, welcher Code für die Verarbeitung der Nachricht zuständig ist. Jeder Prozess verfügt über eine *Mailbox* in der ankommende Nachrichten gepuffert werden. Führt ein Prozess `receive` aus, so wird entweder eine Nachricht aus der *Mailbox* verarbeitet, oder der Prozess wird solange blockiert bis er eine Nachricht empfängt, die mindestens eines der im Konstrukt angegebenen Patterns erfüllt. Außerdem kann mit dem Pattern `after N` Code angegeben werden, den Erlang nach N Sekunden ohne Eintreffen einer passenden Nachricht ausführt.

Damit ein Prozess auf eine Anfrage antworten kann, muss er den Prozess-Identifikator des Absenders kennen. Deshalb ist es Konvention in solchen Fällen ein Tupel zu senden, dessen erstes Element der Prozess-Identifikator des versendenden Prozesses ist. Der Sender einer Nachricht kann seinen eigenen Identifikator mittels der Funktion `self/0` erfragen (siehe Programmausschnitt 7. Ein Beispiel für die Verarbeitung einer solchen Anfrage findet sich in Programmausschnitt 8.

```
server(State) ->
    receive
        {Sender, Query} ->
            {NewState, Reply} =
                reply(Query, State),
            Sender ! Reply,
            server(NewState)
    end.
```

Programmausschnitt 8: Generischer Server, der `reply/2` mit Anfrage und Zustand aufruft und den neuen Zustand und eine Antwort erhält

3.3 Fehlerbehandlung in Aktorsystemen

Da ein verteiltes System eine Voraussetzung für den Bau eines fehlertoleranten Systems ist, muss im Umkehrschluss auch prozessübergreifend auf Fehler reagiert werden können. Zu diesem Zweck können Prozesse miteinander verknüpft werden. Dies geschieht entweder beim Erstellen des Prozesses durch die Verwendung von `spawn_link` anstelle von `spawn` oder nachträglich mittels `link/1`.

Im einfachsten Fall werden dann beim Auftritt eines Fehlers alle mit dem fehlerhaften Prozess verknüpften Prozesse auch terminiert. Kaskadierend werden so alle vom fehlerhaften Prozess abhängigen Prozesse terminiert und können dann neu gestartet werden. Um automatisiert auf das Terminieren von Prozessen zu reagieren bietet Erlang Prozessen die Möglichkeit Fehler von verknüpften Prozessen als gewöhnliche Nachrichten der Form `{'EXIT', Pid, Reason}` zu erhalten. Auf diese Weise lassen sich Kontrollmechanismen implementieren, die beispielsweise Prozesse nach einem Fehler neu starten. Zum Empfang der Nachrichten wird das im vorherigen Abschnitt beschriebene `receive` Konstrukt verwendet. Voraussetzung für den Empfang von Exit-Nachrichten ist, dass der Prozess durch `process_flag(trap_exit, true)` zu einem *System Process* aufgewertet wird. Damit ist kein Prozess des Betriebssystems gemeint; es handelt sich nur um ein interne Unterscheidung Erlangs.

3.4 Open Telecom Platform

Die Open Telecom Platform (OTP) beinhaltet Bibliotheken, Werkzeuge und Abstraktionen (*Behaviours*) von Protokollen für die Entwicklung von fehlertoleranten und verteilten Systemen. *Behaviours* sind generalisierte Anwendungsbestandteile, die mittels Callbacks auf die Anwendung angepasst werden können. Sie sind auf fehlertolerante skalierbare Systeme ausgelegt und unterstützen die dynamische Aktualisierung des Programms zur Laufzeit. Armstrong vergleicht sie in [4] mit J2EE Containern. Laut Larson fördert die Verwendung von *Behaviours* die Robustheit von Programmen, da die Callback-Module ohne *Message-Passing* vollkommen deterministisch werden und somit besser testbar sind [10].

Der mit Abstand am häufigsten eingesetzte *Behaviour* ist der generische Server `gen_server`. Er abstrahiert den bereits vorgestellten Mechanismus zum Austausch von Anfragen und Antworten mit vielen Zusatzfunktionen. Häufig verwendet wer-

den auch der Zustandsautomat `gen_fs`, der Event-Handler `gen_event` und der *Behaviour* zur Überwachung von Prozessen `supervisor`.

4 Zusammenfassung

Die Sprache Erlang eignet sich hervorragend für verteilte Systeme in denen Ausfallsicherheit und Skalierbarkeit von großer Bedeutung sind. Die beschränkte Unterstützung bei der Textverarbeitung und die begrenzten Möglichkeiten benutzerdefinierter Datenstrukturen sorgen aber dafür, dass die nicht objektorientierte Programmiersprache nicht als Allzweckwerkzeug dient. Die klare Zielsetzung bei der Entwicklung hat sich mit der heutigen Vielzahl an verteilten Systemen bezahlt gemacht.

Literatur

[1] J. Armstrong, "A history of erlang," in *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages.* New York, NY, USA: ACM, 2007, pp. 6–1–6–26.

[2] ——, "The development of erlang," in *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming.* New York, NY, USA: ACM, 1997, pp. 196–203.

[3] E. Johansson, M. Pettersson, and K. Sagonas, "A high performance erlang system," in *PPDP '00: Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming.* New York, NY, USA: ACM, 2000, pp. 32–43.

[4] J. Armstrong, "Making Reliable Distributed Systems in the Presence of Software Errors," Ph.D. dissertation, Royal Institute of Technology, Stockholm, Sweden, Nov. 27 2003.

[5] G. Agha, "An overview of actor languages," in *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming.* New York, NY, USA: ACM, 1986, pp. 58–67.

[6] ——, "Computing in pervasive cyberspace," *Commun. ACM*, vol. 51, no. 1, pp. 68–70, 2008.

[7] F. Corrêa, "Actors in a new "highly parallel"world," in *WUP '09: Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010.* New York, NY, USA: ACM, 2009, pp. 21–24.

[8] S. Marr, M. Haupt, and T. D'Hondt, "Intermediate language design of high-level language virtual machines: towards comprehensive concurrency support," in *VMIL '09: Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages.* New York, NY, USA: ACM, 2009, pp. 1–2.

[9] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the jvm platform: a comparative analysis," in *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java.* New York, NY, USA: ACM, 2009, pp. 11–20.

[10] J. Larson, "Erlang for concurrent programming," *Queue*, vol. 6, no. 5, pp. 18–23, 2008.