History of Actors

Tony Garnock-Jones
tonyg@ccs.neu.edu

October 19, 2016

Abstract

Annotated bibliography and talk notes for lecture given on 17 Oct 2016.

Annotated Bibliography

Early work on actors

A universal modular ACTOR formalism for artificial intelligence

```
@inproceedings{Hewitt1973,
  author = {Hewitt, Carl and Bishop, Peter and Steiger, Richard},
  booktitle = {Proc. International Joint Conference on
    Artificial Intelligence},
  pages = {235--245},
  title = {{A universal modular ACTOR formalism for artificial
    intelligence}},
  year = {1973}
}
```

This paper is a position paper from which we can understand the motivation and intentions of the research into actors; it lays out a very broad and colourful research vision that touches on a huge range of areas from computer architecture up through programming language design to teaching of computer science and approaches to artificial intelligence.

The paper presents a *uniform actor model*; compare and contrast with uniform object models offered by (some) OO languages. The original application of the model is given as PLANNER-like AI languages.

The paper claims benefits of using the actor model in a huge range of areas:

- foundations of semantics
- logic
- knowledge-based programming
- intentions (contracts)
- study of expressiveness
- teaching of computation
- extensible, modular programming
- privacy and protection
- synchronization constructs
- resource management
- structured programming
- computer architecture

The paper sketches the idea of a contract (called an "intention") for ensuring that invariants of actors (such as protocol conformance) are maintained; there seems to be a connection to modern work on "monitors" and Session Types. The authors write:

> The intention is the CONTRACT that the actor has with the outside world.

Everything is super meta! Actors can have intentions! Intentions are actors! Intentions can have intentions! The paper presents the beginnings of a reflective metamodel for actors. Every actor has a scheduler and an "intention", and may have monitors, a first-class environment, and a "banker".

The paper draws an explicit connection to capabilities (in the security sense); Mark S. Miller at `http://erights.org/history/actors.html` says of the Actor work that it included "prescient" statements about Actor semantics being "a basis for confinement, years before Norm Hardy and the Key Logic guys", and remarks that "the Key Logic guys were unaware of Actors and the locality laws at the time, [but] were working from the same intuitions."

There are some great slogans scattered throughout, such as "Control flow and data flow are inseparable", and "Global state considered harmful".

The paper does eventually turn to a more nuts-and-bolts description of a predecessor language to PLASMA, which is more fully described in Hewitt 1976.

When it comes to formal reasoning about actor systems, the authors here define a partial order - PRECEDES - that captures some notion of causal connection. Later, the paper makes an excursion into epistemic modal reasoning.

Aside: the paper discusses continuations; Reynolds 1993 has the concept of continuations as firmly part of the discourse by 1973, having been rediscovered in a few different contexts in 1970-71 after van Wijngaarden's 1964 initial description of the idea. See J. C. Reynolds, "The discoveries of continuations," LISP Symb. Comput., vol. 6, no. 3–4, pp. 233–247, 1993.

In the "acknowledgements" section, we see:

> Alan Kay whose FLEX and SMALL TALK machines have influenced our work. Alan emphasized the crucial importance of using intentional definitions of data structures and of passing messages to them. This paper explores the consequences of generalizing the message mechanism of SMALL TALK and SIMULA-67; the port mechanism of Krutar, Balzer, and Mitchell; and the previous CALL statement of PLANNER-71 to a universal communications mechanism.

Semantics of Communicating Parallel Processes

```
@phdthesis{Greif1975,
  author = {Greif, Irene Gloria},
  school = {Massachusetts Institute of Technology},
  title = {{Semantics of Communicating Parallel Processes}},
  type = {PhD dissertation},
  year = {1975}
}
```

Specification language for actors; per Baker an "operational semantics".

Viewing Control Structures as Patterns of Passing Messages

```
@techreport{Hewitt1976,
  author = {Hewitt, Carl},
  number = {AIM-410},
  institution = {MIT Artificial Intelligence Laboratory},
  title = {{Viewing Control Structures as Patterns of Passing Messages}},
  url = {http://dspace.mit.edu/handle/1721.1/6272},
  year = {1976}
}
```

AI focus; actors as "agents" in the AI sense; recursive decomposition: "each of the experts can be viewed as a society that can be further decomposed in the same way until the primitive actors of the system are reached."

> We are investigating the nature of the *communication mechanisms* [...] and the conventions of discourse

More concretely, examines "how actor message passing can be used to understand control structures as patterns of passing messages".

> [...] there is no way to decompose an actor into its parts. An actor is defined by its behavior; not by its physical representation!

Discusses PLASMA ("PLAnner-like System Modeled on Actors"), and gives a fairly detailed description of the language in the appendix. Develops "event diagrams".

Presents very Schemely factorial implementations in recursive and iterative (tail-recursive accumulator-passing) styles. During discussion of the iterative factorial implementation, Hewitt remarks that `n` is not closed over by the `loop` actor; it is "not an acquaintance" of `loop`. Is this the beginning of the line of thinking that led to Clinger's "safe-for-space" work?

Everything is an actor, but some of the actors are treated in an awfully structural way: the trees, for example, in section V.4 on Generators:

```
(non-terminal:
  (non-terminal: (terminal: A) (terminal: B))
  (terminal: C))
```

Things written with this `keyword:` notation look like structures. Their *reflections* are actors, but as structures, they are subject to pattern-matching; I am unsure how the duality here was thought of by the principals at the time, but see the remarks regarding "packages" in the appendix.

Actors and Continuous Functionals

```
@techreport{Hewitt1977,
  author = {Hewitt, Carl and Baker, Henry},
  number = {AIM-436A},
  institution = {MIT Artificial Intelligence Laboratory},
  title = {{Actors and Continuous Functionals}},
  url = {http://hdl.handle.net/1721.1/6687},
  year = {1977}
}
```

Some "laws" for communicating processes; "plausible restrictions on the histories of computations that are physically realizable." Inspired by physical intuition, discusses the history of a computation in terms of a partial order of events, rather than a sequence.

> The actor model is a formalization of these ideas [of Simula/Smalltalk/CLU-like active data processing messages] that is independent of any particular programming language.

> Instances of Simula and Smalltalk classes and CLU clusters are actors", but they are *non-concurrent*. The actor model is broader, including concurrent message-passing behaviour.

Laws about (essentially) lexical scope. Laws about histories (finitude of activation chains; total ordering of messages inbound at an actor; etc.), including four different ordering relations. "Laws of locality" are what Miller was referring to on that erights.org page I mentioned above; very close to the capability laws governing confinement.

Steps toward denotational semantics of actors.

Synchronization in actor systems

```
@article{Atkinson1977,
  author = {Atkinson, R. and Hewitt, Carl},
  journal = {Proc. Symp. on Principles of Programming Languages},
  title = {{Synchronization in actor systems}},
  url = {http://portal.acm.org/citation.cfm?id=512975},
  year = {1977}
}
```

Introduces the concept of "serializers", a "generalization and improvement of the monitor mechanism of Brinch-Hansen and Hoare". Builds on Greif's work.

Foundations of Actor Semantics

```
@phdthesis{Clinger1981,
  author = {Clinger, William Douglas},
  school = {Massachusetts Institute of Technology},
  title = {{Foundations of Actor Semantics}},
  type = {PhD dissertation},
  year = {1981}
}
```

Actors as Concurrent Object-Oriented Programming

Actors: a model of concurrent computation in distributed systems

```
@book{Agha1986,
  author = {Agha, Gul},
  isbn = {0-262-01092-5},
  publisher = {MIT Press},
  title = {{Actors: a model of concurrent computation in distributed systems}},
  year = {1986}
}
```

Concurrent Object-Oriented Programming

```
@article{Agha1990,
  author = {Agha, Gul},
  journal = {Communications of the ACM},
  number = {9},
  pages = {125--141},
  title = {{Concurrent Object-Oriented Programming}},
  volume = {33},
  year = {1990}
}
```

Agha's work recast the early "classic actor model" work in terms of *concurrent object-oriented programming*. Here, he defines actors as "self-contained, interactive, independent components of a computing system that communicate by asynchronous message passing", and gives the basic actor primitives as `create`, `send to`, and `become`. Examples are given in the actor language Rosette.

This paper gives an overview and summary of many of the important facets of research on actors that had been done at the time, including brief discussion of: nondeterminism and fairness; patterns of coordination beyond simple request/reply such as transactions; visualization, monitoring and debugging; resource management in cases of extremely high levels of potential concurrency; and reflection.

> The customer-passing style supported by actors is the concurrent generalization of continuation-passing style supported in sequential languages such as Scheme. In case of sequential systems, the object must have completed processing a communication before it can process another communication. By contrast, in concurrent systems it is possible to process the next communication as soon as the replacement behavior for an object is known.

Note that the sequential-seeming portions of the language are defined in terms of asynchronous message passing and construction of explicit continuation actors.

A Foundation for Actor Computation

```
@article{Agha1997,
  author = {Agha, Gul A. and Mason, Ian A. and Smith, Scott F. and Talcott, Carolyn L.},
  journal = {Journal of Functional Programming},
  number = {1},
  pages = {1--72},
  title = {{A Foundation for Actor Computation}},
  volume = {7},
  year = {1997}
}
```

Long paper that carefully and fully develops an operational semantics for a concrete actor language based on lambda-calculus. Discusses various equivalences and laws. An excellent starting point if you're looking to build on a modern approach to operational semantics for actors.

Erlang: Actors from requirements for fault-tolerance / high-availability

Making reliable distributed systems in the presence of software errors

```
@phdthesis{Armstrong2003,
  author = {Armstrong, Joe},
  school = {Royal Institute of Technology, Stockholm},
  title = {{Making reliable distributed systems in the presence of software errors}},
  type = {PhD dissertation},
  year = {2003}
}
```

A good overview of Erlang: the language, its design intent, and the underlying philosophy. Includes an evaluation of the language design.

E: Actors from requirements for secure interaction

Concurrency Among Strangers

```
@inproceedings{Miller2005,
  author = {Miller, Mark S. and Tribble, E. Dean and Shapiro, Jonathan},
  booktitle = {Proc. Int. Symp. on Trustworthy Global Computing},
  pages = {195--229},
  title = {{Concurrency Among Strangers}},
  year = {2005}
}
```

As I summarised this paper for a seminar class on distributed systems: "The authors present E, a language designed to help programmers manage *coordination* of concurrent activities in a setting of distributed, mutually-suspicious objects. The design features of E allow programmers to take control over concerns relevant to distributed systems, without immediately losing the benefits of ordinary OO programming."

E is a canonical example of the "communicating event loops" approach to Actor languages, per the taxonomy of the survey paper listed below. It combines message-passing and isolation in an interesting way with ordinary object-oriented programming, giving a two-level language structure that has an OO flavour.

The paper does a great job of explaining the difficulties that arise when writing concurrent programs in traditional models, thereby motivating the actor model in general and the features of E in particular as a way of making the programmer's job easier.

Taxonomy of actors

43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties

```
@techreport{DeKoster2016,
  author = {{De Koster}, Joeri and {Van Cutsem}, Tom and {De Meuter}, Wolfgang},
  institution = {Software Languages Lab, Vrije Universiteit Brussel},
  title = {{43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties}},
  year = {2016}
}
```

A very recent survey paper (the BibTeX entry just above refers to the tech report, but the paper is to appear at SPLASH AGERE 2016) that offers a taxonomy for classifying actor-style languages. At its broadest, actor languages are placed in one of four groups:

- The Classic Actor Model (create, send, become)
- Active Objects (OO with a thread per object; copying of passive data between objects)
- Processes (raw Erlang; receive, spawn, send)
- Communicating Event-Loops (E; near and far references; eventual references; batching)

Different kinds of "futures" or "promises" also appear in many of these variations in order to integrate asynchronous message *reception* with otherwise-sequential programming.

Talk Notes: History of Actors

Introduction

Today I'm going to talk about the actor model. I'll first put the model in context, and then show three different styles of actor language, including two that aim to be realistic programming systems.

I'm going to draw on a few papers:

- 2016 survey by Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter, for taxonomy and common terminology (SPLASH AGERE 2016)
- 1990 overview by Gul Agha, who has contributed hugely to the study of actors (Comm. ACM 1990)
- 1997 operational semantics for actors by Gul Agha, Ian Mason, Scott Smith, and Carolyn Talcott.
- brief mention of some of the content of the original 1973 paper by Carl Hewitt, Peter Bishop, and Richard Steiger. (IJCAI 1973)
- Joe Armstrong's 2003 dissertation on Erlang.
- 2005 paper on the language E by Mark Miller, Dean Tribble, and Jonathan Shapiro. (TGC 2005)

Actors in context: Approaches to concurrent programming

One way of classifying approaches is along a spectrum of private vs. shared state.

- Shared memory, threads, and locking: very limited private state, almost all shared
- Tuple spaces and my own research, Syndicate: some shared, some private and isolated
- Actors: almost all private and isolated, just enough shared to do routing

Pure functional "data parallelism" doesn't fit on this chart - it lacks shared mutable state entirely.

The actor model

The actor model, then is

- asynchronous message passing between entities (actors)
  - with guaranteed delivery

  - addressing of messages by actor identity
- a thing called the "ISOLATED TURN PRINCIPLE"
  - no shared mutable state; strong encapsulation; no global mutable state
  - no interleaving; process one message at a time; serializes state access
  - liveness; no blocking

What it buys you

- modular reasoning about state in the overall system, and modular reasoning about local state change within an actor, because state is private, and access is serialized; only have to consider interleavings of messages
- no "deadlocks" or data races (though you can get "datalock" and other global non-progress in some circumstances, from logical inconsistency and otherwise)
- flexible, powerful capability-based security
- failure isolation and fault tolerance

It's worth remarking that actor-like concepts have sprung up several times independently. Hewitt and many others invented and developed actors in the 1970s, but there are two occasions where actors seem to have been independently reinvented, as far as I know.

One is work on a capability-based operating system, KeyKOS, in the 1980s which involved a design very much like Hewitt's actors, feeding into research which led ultimately to the language E.

The other is work on highly fault-tolerant designs for telephone switches, also in the 1980s, which culminated in the language Erlang.

Both languages are clearly actor languages, and in both cases, apparently the people involved were unaware of Hewitt's actor model at the time.

Terminology

There are two kinds of things in the actor model: *messages*, which are data sent across some medium of communication, and *actors*, which are stateful entities that can only affect each other by sending messages back and forth.

Messages are completely immutable data, passed by copy, which may contain references to other actors.

Each actor has

- private state; analogous to instance variables

- an interface; which messages it can respond to

Together, the private state and the interface make up the actor's *behaviour*, a key term in the actor literature.

In addition, each actor has

- a mailbox; inbox, message queue
- an address; denotes the mailbox

Within this framework, there has been quite a bit of variety in how the model appears as a concrete programming language.

De Koster et al. classify actor languages into

- The Classic Actor Model (create, send, become)
- Active Objects (OO with a thread per object; copying of passive data between objects)
- Processes (raw Erlang; receive, spawn, send)
- Communicating Event-Loops (no passive data; E; near/far refs; eventual refs; batching)

Rest of the talk

For the rest of the talk, I'm going to cover the classic actor model using Agha's presentations as a guide; then I'll compare it to E, a communicating event-loop actor language, and to Erlang, a process actor language.

Classic Actor Model

The original 1973 actor paper by Hewitt, Bishop and Steiger in the International Joint Conference on Artificial Intelligence, is incredibly far out!

It's a position paper that lays out a broad and colourful research vision. It's packed with amazing ideas.

The heart of it is that Actors are proposed as a universal programming language formalism ideally suited to building artificial intelligence.

The goal really was A.I., and actors and programming languages were a means to that end.

It makes these claims that actors bring great benefits in a huge range of areas:

- foundations of semantics
- logic
- knowledge-based programming
- intentions (software contracts)
- study of expressiveness of programming languages
- teaching of computation
- extensible, modular programming
- privacy and protection
- synchronization constructs
- resource management
- structured programming
- computer architecture

(It's amazingly "full-stack": computer architecture!?)

In each of these areas, you can see what they were going for. In some, actors have definitely been useful; in others, the results have been much more modest.

In the mid-to-late 70s, Hewitt and his students Irene Greif, Henry Baker, and Will Clinger developed a lot of the basic theory of the actor model, inspired originally by SIMULA and Smalltalk-71. Irene Greif developed the first operational semantics for it as her dissertation work and Will Clinger developed a denotational semantics for actors.

In the late 70s through the 80s and beyond, Gul Agha made huge contributions to the actor theory. His dissertation was published as a book on actors in 1986 and has been very influential. He separated the actor model from its A.I. roots and started treating it as a more general programming model. In particular, in his 1990 Comm. ACM paper, he describes it as a foundation for CONCURRENT OBJECT-ORIENTED PROGRAMMING.

Agha's formulation is based around the three core operations of the classic actor model:

- `create`: constructs a new actor from a template and some parameters
- `send`: delivers a message, asynchronously, to a named actor
- `become`: within an actor, replaces its behaviour (its state & interface)

The classic actor model is a UNIFORM ACTOR MODEL, that is, everything is an actor. Compare to uniform object models, where everything is an object. By the mid-90s, that very strict uniformity had fallen out of favour and people often worked with *two-layer* languages, where you might have a functional core language, or an object-oriented core language, or an imperative core language with the actor model part being added in to the base language.

I'm going to give a simplified, somewhat informal semantics based on his 1997 work with Mason, Smith and Talcott. I'm going to drop a lot of details that aren't relevant here so this really will be simplified.

```
e  :=  \x.e  |  e e  |  x  | (e,e) | l
    | ... atoms, if, bool, primitive operations ...
    |  create e
    |  send e e
    |  become e

l  labels, PIDs; we'll use them like symbols here
```

and we imagine convenience syntax

```
e1;e2              to stand for   (\dummy.e2) e1
let x = e1 in e2   to stand for   (\x.e2) e1

match e with p -> e, p -> e, ...
    to stand for matching implemented with if, predicates, etc.
```

We forbid programs from containing literal process IDs.

```
v  :=  \x.e  |  (v,v)  |  l

R  :=  []  |  R e  |  v R  |  (R,e)  |  (v,R)
    |  create R
    |  send R e  |  send v R
    |  become R
```

Configurations are a pair of a set of actors and a multiset of messages:

```
m  :=  l <-- v

A  :=  (v)l  |  [e]l

C  :=  < A ... | m ... >
```

The normal lambda-calculus-like reductions apply, like beta:

```
    < ... [R[(\x.e) v]]l ... | ... >
--> < ... [R[e{v/x}  ]]l ... | ... >
```

Plus some new interesting ones that are actor specific:

```
    < ... (v)l ...    | ... l <-- v' ... >
--> < ... [v v']l ... | ...          ... >

    < ... [R[create v]]l       ... | ... >
```

```
--> < ... [R[l'      ]]l (v)l' ... | ... >   where l' fresh

    < ... [R[send l' v]]l ... | ... >
--> < ... [R[l'       ]]l ... | ... l' <-- v >

    < ... [R[become v]]l       ... | ... >
--> < ... [R[nil     ]]l' (v)l ... | ... >   where l' fresh
```

Whole programs `e` are started with

```
< [e]a | >
```

where `a` is an arbitrary label.

Here's an example - a mutable cell.

```
Cell = \contents . \message . match message with
                                (get, k) --> become (Cell contents);
                                             send k contents
                                (put, v) --> become (Cell v)
```

Notice that when it gets a `get` message, it first performs a `become` in order to quickly return to `ready` state to handle more messages. The remainder of the code then runs alongside the ready actor. Actions after a `become` can't directly affect the state of the actor anymore, so even though we have what looks like multiple concurrent executions of the actor, there's no sharing, and so access to the state is still serialized, as needed for the isolated turn principle.

```
< [let c = create (Cell 0) in
   send c (get, create (\v . send c (put, v + 1)))]a | >

< [let c = l1 in
   send c (get, create (\v . send c (put, v + 1)))]a
  (Cell 0)l1 | >

< [send l1 (get, create (\v . send l1 (put, v + 1)))]a
  (Cell 0)l1 | >

< [send l1 (get, l2)]a
  (Cell 0)l1
  (\v . send l1 (put, v + 1))l2 | >

< [l1]a
  (Cell 0)l1
  (\v . send l1 (put, v + 1))l2 | l1 <-- (get, l2) >
```

```
< [l1]a
  [Cell 0 (get, l2)]l1
  (\v . send l1 (put, v + 1))l2 | >

< [l1]a
  [become (Cell 0); send l2 0]l1
  (\v . send l1 (put, v + 1))l2 | >

< [l1]a
  [send l2 0]l3
  (Cell 0)l1
  (\v . send l1 (put, v + 1))l2 | >

< [l1]a
  [l2]l3
  (Cell 0)l1
  (\v . send l1 (put, v + 1))l2 | l2 <-- 0 >

< [l1]a
  [l2]l3
  (Cell 0)l1
  [send l1 (put, 0 + 1)]l2 | >

< [l1]a
  [l2]l3
  (Cell 0)l1
  [l1]l2 | l1 <-- (put, 1) >

< [l1]a
  [l2]l3
  [Cell 0 (put, 1)]l1
  [l1]l2 | >

< [l1]a
  [l2]l3
  [become (Cell 1)]l1
  [l1]l2 | >

< [l1]a
  [l2]l3
  [nil]l4
  (Cell 1)l1
  [l1]l2 | >
```

(You could consider adding a garbage collection rule like

```
    < ... [v]l ... | ... >
--> < ...      ... | ... >
```

to discard the final value at the end of an activation.)

Because at this level all the continuations are explicit, you can encode patterns other than sequential control flow, such as fork-join.

For example, to start two long-running computations in parallel, and collect the answers in either order, multiplying them and sending the result to some actor k', you could write

```
let k = create (\v1 . become (\v2 . send k' (v1 * v2))) in
send task1 (req1, k);
send task2 (req2, k)
```

Practically speaking, both Hewitt's original actor language, PLASMA, and the language Agha uses for his examples in the 1990 paper, Rosette, have special syntax for ordinary RPC so the programmer needn't manipulate continuations themselves.

So that covers the classic actor model. Create, send and become. Explicit use of actor addresses, and lots and lots of temporary actors for inter-actor RPC continuations.

Before I move on to Erlang: remember right at the beginning I told you the actor model was

- asynchronous message passing, and
- the isolated turn principle

?

The isolated turn principle requires *liveness* - you're not allowed to block indefinitely while responding to a message!

But here, we can:

```
let d = create (\c . send c c) in
send d d
```

Compare this with

```
letrec beh = (\c . become beh; send c c) in
let d = create beh in
send d d
```

These are both degenerate cases, but in different ways: the first becomes inert very quickly and the actor `d` is never returned to an idle/ready state, while the second spins uselessly forever.

Other errors we could make would be to fail to send an expected reply to a continuation.

Even if we require our behaviour functions to be total, we can still get *global* nontermination.

So saying that we "don't have deadlock" with the actor model is very much oversimplified, even at the level of simple formal models, let alone when it comes to realistic programming systems.

Erlang: Actors for fault-tolerant systems

Erlang is an example of what De Koster et al. call a Process-based actor language.

It has its origins in telephony, where it has been used to build telephone switches with fabled "nine nines" of uptime. The research process that led to Erlang concentrated on high-availability, fault-tolerant software. The reasoning that led to such an actor-like system was, in a nutshell:

- Programs have bugs. If part of the program crashes, it shouldn't corrupt other parts. Hence strong isolation and shared-nothing message-passing.
- If part of the program crashes, another part has to take up the slack, one way or another. So we need crash signalling so we can detect failures and take some action.
- We can't have all our eggs in one basket! If one machine fails at the hardware level, we need to have a nearby neighbour that can smoothly continue running. For that redundancy, we need distribution, which makes the shared-nothing message passing extra attractive as a unifying mechanism.

Erlang is a two-level system, with a functional language core equipped with imperative actions for asynchronous message send and spawning new processes, like Agha's system.

The difference is that it lacks `become`, and instead has a construct called `receive`.

Erlang actors, called processes, are ultra lightweight threads that run sequentially from beginning to end as little functional programs. As it runs, no explicit temporary continuation actors are created: any time it uses receive, it simply blocks until a matching message appears.

After some initialization steps, these programs typically enter a message loop. For example, here's a mutable cell:

```
mainloop(Contents) ->
  receive
    {get, K} -> K ! Contents,
                mainloop(Contents);
```

```
    {put, V} -> mainloop(V)
  end.
```

A client program might be

```
Cell = spawn(fun mainloop(0) end),
Cell ! {get, self()},
receive
  V -> ...
end.
```

Instead of using `become`, the program performs a tail call which returns to the `receive` statement as the last thing it does.

Because `receive` is a statement like any other, Erlang processes can use it to enter substates:

```
mainloop(Service) ->
  receive
    {req, K} -> Service ! {subreq, self()},
                receive
                  {subreply, Answer} -> K ! {reply, Answer},
                                        mainloop(Service)
                end
  end.
```

While the process is blocked on the inner `receive`, it only processes messages matching the patterns in that inner receive, and it isn't until it does the tail call back to `mainloop` that it starts waiting for `req` messages again. In the meantime, non-matched messages queue up waiting to be `receive`d later.

This is called "selective receive" and it is difficult to reason about. However, the underlying goal of changing the set of messages one responds to and temporarily ignoring others is important for the way people think about actor systems, and a lot of research has been done on different ways of selectively enabling message handlers. See Agha's 1990 paper for pointers toward this research.

One unique feature that Erlang brings to the table is crash signalling. The jargon is "links" and "monitors". Processes can ask the system to make sure to send them a message if a monitored process exits. That way, they can perform RPC by

- monitoring the server process
- sending the request
- if a reply message arrives, unmonitor the server process and continue
- if an exit message arrives, the service has crashed; take some corrective action.

This general idea of being able to monitor the status of some other process was one of the seeds of my own research and my language Syndicate.

So while the classic actor model had create/send/become as primitives, Erlang has spawn/send/receive, and actors are processes rather than event-handler functions. The programmer still manipulates references to actors/processes directly, but there are far fewer explicit temporary actors created compared to the "classic model"; the ordinary continuations of Erlang's functional fragment take on those duties.

E: actors for secure cooperation

The last language I want to show you, E, is an example of what De Koster et al. call a Communicating Event-Loop language.

E looks and feels much more like a traditional object-oriented language to the programmer than either of the variations we've seen so far.

```
def makeCell (var contents) {
    def getter {
        to get() { return contents }
    }
    def setter {
        to set(newContents) {
            contents := newContents
        }
    }
    return [getter, setter]
}
```

E uses the term "vat" to describe the concept closest to the traditional actor. A vat has not only a mailbox, like an actor, but also a call stack, and a heap of local objects. As we'll see, E programmers don't manipulate references to vats directly. Instead, they pass around references to objects in a vat's heap.

There are two kinds of references available: near refs, which definitely point to objects in the local vat, and far refs, which may point to objects in a different vat, perhaps on another machine.

To go with these two kinds of refs, there are two kinds of method calls: *immediate* calls, and *eventual* calls.

receiver.method(arg, . . .)

receiver <- method(arg, . . .)

It is an error to use an immediate call on a ref to an object in a different vat, because it blocks during the current turn while the answer is computed. It's OK to use eventual

calls on any ref at all, though: it causes a message to be queued in the target vat (which might be our own), and a *promise* is immediately returned to the caller.

The promise starts off unresolved. Later, when the target vat has computed and sent a reply, the promise will become resolved. A nifty trick is that even an unresolved promise is useful: you can *pipeline* them. For example,

```
def r1 := x <- a()
def r2 := y <- b()
def r3 := r1 <- c(r2)
```

would block and perform multiple network round trips in a traditional simple RPC system; in E, there is a protocol layered on top of raw message sending that discusses promise creation, resolution and use. This protocol allows the system to send messages like

```
"Send a() to x, and name the resulting promise r1"
"Send b() to y, and name the resulting promise r2"
"When r1 is known, send c(r2) to it"
```

Crucial here is that the protocol, the language of discourse between actors, allows the expression of concepts including the notion of a send to happen at a future time, to a currently-unknown recipient.

The protocol and the E vat runtimes work together to make sure that messages get to where they need to go efficiently, even in the face of multiple layers of forwarding.

Each turn of an E vat involves taking one message off the message queue, and dispatching it to the local object it denotes. Immediate calls push stack frames on the stack as usual for object-oriented programming languages; eventual calls push messages onto the message queue. Execution continues until the stack is empty again, at which point the turn concludes and the turn starts.

Turning to failure signalling, the E approach is quite different to the Erlang approach. In E, the programmer has a convenient value that represents the outcome of a transaction: the promise. When an exception is signalled, or a network problem arises, any related promises are put into a special state: they become *broken promises*. Interacting with a broken promise causes the contained exception to be signalled; in this way, broken promises propagate failure along causal chains.

If we look under the covers, E seems to have a "classic style" model using create/send/become to manage and communicate between whole vats, but these operations aren't exposed to the programmer. The programmer instead manipulates two-part "far references" which denote a vat along with an object local to that vat. Local objects are created frequently, like in regular object-oriented languages, but vats are created much less frequently; and each vat's stack takes on duties performed in "classic" actor models by temporary actors.

Conclusion

I've presented three different types of actor language: the classic actor model, roughly as formulated by Agha et al.; the process actor model, represented by Erlang; and the communicating event-loop model, represented by E.

The three models take different approaches to reconciling the need to have structured local data within each actor in addition to the more coarse-grained structure relating actors to each other.

The classic model makes everything an actor, with local data largely deemphasised; Erlang offers a traditional functional programming model for handling local data; and E offers a smooth integration between an imperative local OO model and an asynchronous, promise-based remote OO model.