WIKIPEDIA

Actor model

The **actor model** in computer science is a mathematical model of concurrent computation that treats *actor* as the universal primitive of concurrent computation. In response to a message it receives, an actor can: make local decisions, create more actors, send more messages, and determine how to respond to the next message received. Actors may modify their own private state, but can only affect each other indirectly through messaging (removing the need for lock-based synchronization).

The actor model originated in 1973.[1] It has been used both as a framework for a theoretical understanding of computation and as the theoretical basis for several practical implementations of concurrent systems. The relationship of the model to other work is discussed in actor model and process calculi.

Contents

History

Fundamental concepts

Formal systems

Applications

Message-passing semantics
- Unbounded nondeterminism controversy
- Direct communication and asynchrony
- Actor creation plus addresses in messages means variable topology
- Inherently concurrent
- No requirement on order of message arrival
- Locality
- Composing actor systems
- Behaviors
- Modeling other concurrency systems
- Computational Representation Theorem
- Relationship to logic programming
- Migration
- Security
- Synthesizing addresses of actors
- Contrast with other models of message-passing concurrency

Influence
- Theory
- Practice

Addressed issues

Programming with actors
- Early actor programming languages
- Later actor programming languages
- Actor libraries and frameworks

See also
References
Further reading
External links

History

According to Carl Hewitt, unlike previous models of computation, the actor model was inspired by physics, including general relativity and quantum mechanics. It was also influenced by the programming languages Lisp, Simula, early versions of Smalltalk, capability-based systems, and packet switching. Its development was "motivated by the prospect of highly parallel computing machines consisting of dozens, hundreds, or even thousands of independent microprocessors, each with its own local memory and communications processor, communicating via a high-performance communications network."[2] Since that time, the advent of massive concurrency through multi-core and manycore computer architectures has revived interest in the actor model.

Following Hewitt, Bishop, and Steiger's 1973 publication, Irene Greif developed an operational semantics for the actor model as part of her doctoral research.[3] Two years later, Henry Baker and Hewitt published a set of axiomatic laws for actor systems.[4][5] Other major milestones include William Clinger's 1981 dissertation introducing a denotational semantics based on power domains[2] and Gul Agha's 1985 dissertation which further developed a transition-based semantic model complementary to Clinger's.[6] This resulted in the full development of actor model theory.

Major software implementation work was done by Russ Atkinson, Giuseppe Attardi, Henry Baker, Gerry Barber, Peter Bishop, Peter de Jong, Ken Kahn, Henry Lieberman, Carl Manning, Tom Reinhardt, Richard Steiger and Dan Theriault in the Message Passing Semantics Group at Massachusetts Institute of Technology (MIT). Research groups led by Chuck Seitz at California Institute of Technology (Caltech) and Bill Dally at MIT constructed computer architectures that further developed the message passing in the model. See Actor model implementation.

Research on the actor model has been carried out at California Institute of Technology, Kyoto University Tokoro Laboratory, Microelectronics and Computer Technology Corporation (MCC), MIT Artificial Intelligence Laboratory, SRI, Stanford University, University of Illinois at Urbana–Champaign,[7] Pierre and Marie Curie University (University of Paris 6), University of Pisa, University of Tokyo Yonezawa Laboratory, Centrum Wiskunde & Informatica (CWI) and elsewhere.

Fundamental concepts

The actor model adopts the philosophy that *everything is an actor*. This is similar to the *everything is an object* philosophy used by some object-oriented programming languages.

An actor is a computational entity that, in response to a message it receives, can concurrently:

- send a finite number of messages to other actors;
- create a finite number of new actors;
- designate the behavior to be used for the next message it receives.

There is no assumed sequence to the above actions and they could be carried out in parallel.

Decoupling the sender from communications sent was a fundamental advance of the actor model enabling asynchronous communication and control structures as patterns of passing messages.[8]

Recipients of messages are identified by address, sometimes called "mailing address". Thus an actor can only communicate with actors whose addresses it has. It can obtain those from a message it receives, or if the address is for an actor it has itself created.

The actor model is characterized by inherent concurrency of computation within and among actors, dynamic creation of actors, inclusion of actor addresses in messages, and interaction only through direct asynchronous message passing with no restriction on message arrival order.

Formal systems

Over the years, several different formal systems have been developed which permit reasoning about systems in the actor model. These include:

- Operational semantics[3][9]
- Laws for actor systems[4]
- Denotational semantics[2][10]
- Transition semantics[6]

There are also formalisms that are not fully faithful to the actor model in that they do not formalize the guaranteed delivery of messages including the following (See Attempts to relate actor semantics to algebra and linear logic):

- Several different actor algebras[11][12][13]
- Linear logic[14]

Applications

The actor model can be used as a framework for modeling, understanding, and reasoning about a wide range of concurrent systems.[15] For example:

- Electronic mail (email) can be modeled as an actor system. Accounts are modeled as actors and email addresses as actor addresses.
- Web services can be modeled with Simple Object Access Protocol (SOAP) endpoints modeled as actor addresses.
- Objects with locks (*e.g.*, as in Java and C#) can be modeled as a *serializer*, provided that their implementations are such that messages can continually arrive (perhaps by being stored in an internal queue). A serializer is an important kind of actor defined by the property that it is continually available to the arrival of new messages; every message sent to a serializer is guaranteed to arrive.[16]
- Testing and Test Control Notation (TTCN), both TTCN-2 and TTCN-3, follows actor model rather closely. In TTCN actor is a test component: either parallel test component (PTC) or main test component (MTC). Test components can send and receive messages to and from remote partners (peer test components or test system interface), the latter being identified by its address. Each test component has a behaviour tree bound to it; test components run in parallel and can be dynamically created by parent test components. Built-in language constructs allow the definition of actions to be taken when an expected message is received

from the internal message queue, like sending a message to another peer entity or creating new test components.

Message-passing semantics

The actor model is about the semantics of message passing.

Unbounded nondeterminism controversy

Arguably, the first concurrent programs were interrupt handlers. During the course of its normal operation a computer needed to be able to receive information from outside (characters from a keyboard, packets from a network, *etc*). So when the information arrived the execution of the computer was *interrupted* and special code (called an interrupt handler) was called to put the information in a data buffer where it could be subsequently retrieved.

In the early 1960s, interrupts began to be used to simulate the concurrent execution of several programs on one processor.[17] Having concurrency with shared memory gave rise to the problem of concurrency control. Originally, this problem was conceived as being one of mutual exclusion on a single computer. Edsger Dijkstra developed semaphores and later, between 1971 and 1973,[18] Tony Hoare[19] and Per Brinch Hansen[20] developed monitors to solve the mutual exclusion problem. However, neither of these solutions provided a programming language construct that encapsulated access to shared resources. This encapsulation was later accomplished by the serializer construct ([Hewitt and Atkinson 1977, 1979] and [Atkinson 1980]).

The first models of computation (*e.g.*, Turing machines, Post productions, the lambda calculus, *etc.*) were based on mathematics and made use of a global state to represent a computational *step* (later generalized in [McCarthy and Hayes 1969] and [Dijkstra 1976] see Event orderings versus global state). Each computational step was from one global state of the computation to the next global state. The global state approach was continued in automata theory for finite-state machines and push down stack machines, including their nondeterministic versions. Such nondeterministic automata have the property of bounded nondeterminism; that is, if a machine always halts when started in its initial state, then there is a bound on the number of states in which it halts.

Edsger Dijkstra further developed the nondeterministic global state approach. Dijkstra's model gave rise to a controversy concerning unbounded nondeterminism (also called *unbounded indeterminacy*), a property of concurrency by which the amount of delay in servicing a request can become unbounded as a result of arbitration of contention for shared resources *while still guaranteeing that the request will eventually be serviced*. Hewitt argued that the actor model should provide the guarantee of service. In Dijkstra's model, although there could be an unbounded amount of time between the execution of sequential instructions on a computer, a (parallel) program that started out in a well defined state could terminate in only a bounded number of states [Dijkstra 1976]. Consequently, his model could not provide the guarantee of service. Dijkstra argued that it was impossible to implement unbounded nondeterminism.

Hewitt argued otherwise: there is no bound that can be placed on how long it takes a computational circuit called an *arbiter* to settle (see metastability (electronics)).[21] Arbiters are used in computers to deal with the circumstance that computer clocks operate asynchronously with respect to input from outside, *e.g.*, keyboard input, disk access, network input, *etc.* So it could take an unbounded time for a message sent to a computer to be received and in the meantime the computer could traverse an unbounded number of states.

The actor model features unbounded nondeterminism which was captured in a mathematical model by Will Clinger using domain theory.[2] In the actor model, there is no global state.

Direct communication and asynchrony

Messages in the actor model are not necessarily buffered. This was a sharp break with previous approaches to models of concurrent computation. The lack of buffering caused a great deal of misunderstanding at the time of the development of the actor model and is still a controversial issue. Some researchers argued that the messages are buffered in the "ether" or the "environment". Also, messages in the actor model are simply sent (like packets in IP); there is no requirement for a synchronous handshake with the recipient.

Actor creation plus addresses in messages means variable topology

A natural development of the actor model was to allow addresses in messages. Influenced by packet switched networks [1961 and 1964], Hewitt proposed the development of a new model of concurrent computation in which communications would not have any required fields at all: they could be empty. Of course, if the sender of a communication desired a recipient to have access to addresses which the recipient did not already have, the address would have to be sent in the communication.

For example, an actor might need to send a message to a recipient actor from which it later expects to receive a response, but the response will actually be handled by a third actor component that has been configured to receive and handle the response (for example, a different actor implementing the observer pattern). The original actor could accomplish this by sending a communication that includes the message it wishes to send, along with the address of the third actor that will handle the response. This third actor that will handle the response is called the *resumption* (sometimes also called a continuation or stack frame). When the recipient actor is ready to send a response, it sends the response message to the *resumption* actor address that was included in the original communication.

So, the ability of actors to create new actors with which they can exchange communications, along with the ability to include the addresses of other actors in messages, gives actors the ability to create and participate in arbitrarily variable topological relationships with one another, much as the objects in Simula and other object-oriented languages may also be relationally composed into variable topologies of message-exchanging objects.

Inherently concurrent

As opposed to the previous approach based on composing sequential processes, the actor model was developed as an inherently concurrent model. In the actor model sequentiality was a special case that derived from concurrent computation as explained in actor model theory.

No requirement on order of message arrival

Hewitt argued against adding the requirement that messages must arrive in the order in which they are sent to the actor. If output message ordering is desired, then it can be modeled by a queue actor that provides this functionality. Such a queue actor would queue the messages that arrived so that they could be retrieved in FIFO order. So if an actor `X` sent a message `M1` to an actor `Y`, and later `X` sent another message `M2` to `Y`, there is no requirement that `M1` arrives at `Y` before `M2`.

In this respect the actor model mirrors packet switching systems which do not guarantee that packets must be received in the order sent. Not providing the order of delivery guarantee allows packet switching to buffer packets, use multiple paths to send packets, resend damaged packets, and to provide other optimizations.

For example, actors are allowed to pipeline the processing of messages. What this means is that in the course of processing a message `M1`, an actor can designate the behavior to be used to process the next message, and then in fact begin processing another message `M2` before it has finished processing `M1`. Just because an actor is allowed to pipeline the processing of messages does not mean that it *must* pipeline the processing. Whether a message is pipelined is an engineering tradeoff. How would an external observer know whether the processing of a message by an actor has been pipelined? There is no ambiguity in the definition of an actor created by the possibility of pipelining. Of course, it is possible to perform the pipeline optimization incorrectly in some implementations, in which case unexpected behavior may occur.

Locality

Another important characteristic of the actor model is locality.

Locality means that in processing a message, an actor can send messages only to addresses that it receives in the message, addresses that it already had before it received the message, and addresses for actors that it creates while processing the message. (But see Synthesizing addresses of actors.)

Also locality means that there is no simultaneous change in multiple locations. In this way it differs from some other models of concurrency, *e.g.*, the Petri net model in which tokens are simultaneously removed from multiple locations and placed in other locations.

Composing actor systems

The idea of composing actor systems into larger ones is an important aspect of modularity that was developed in Gul Agha's doctoral dissertation,[6] developed later by Gul Agha, Ian Mason, Scott Smith, and Carolyn Talcott.[9]

Behaviors

A key innovation was the introduction of *behavior* specified as a mathematical function to express what an actor does when it processes a message, including specifying a new behavior to process the next message that arrives. Behaviors provided a mechanism to mathematically model the sharing in concurrency.

Behaviors also freed the actor model from implementation details, *e.g.*, the Smalltalk-72 token stream interpreter. However, it is critical to understand that the efficient implementation of systems described by the actor model require *extensive* optimization. See Actor model implementation for details.

Modeling other concurrency systems

Other concurrency systems (*e.g.*, process calculi) can be modeled in the actor model using a two-phase commit protocol.[22]

Computational Representation Theorem

There is a *Computational Representation Theorem* in the actor model for systems which are closed in the sense that they do not receive communications from outside. The mathematical denotation denoted by a closed system **S** is constructed from an initial behavior $\perp_S$ and a behavior-approximating function

$\mathbf{progression}_S$. These obtain increasingly better approximations and construct a denotation (meaning) for **S** as follows [Hewitt 2008; Clinger 1981]:

$$\mathbf{Denote_S} \equiv \lim_{i \to \infty} \mathbf{progression_{S^i}}(\bot_S)$$

In this way, S can be mathematically characterized in terms of all its possible behaviors (including those involving unbounded nondeterminism). Although $\mathbf{Denote_S}$ is not an implementation of **S**, it can be used to prove a generalization of the Church-Turing-Rosser-Kleene thesis [Kleene 1943]:

A consequence of the above theorem is that a finite actor can nondeterministically respond with an uncountable number of different outputs.

Relationship to logic programming

One of the key motivations for the development of the actor model was to understand and deal with the control structure issues that arose in development of the Planner programming language. Once the actor model was initially defined, an important challenge was to understand the power of the model relative to Robert Kowalski's thesis that "computation can be subsumed by deduction". Hewitt argued that Kowalski's thesis turned out to be false for the concurrent computation in the actor model (see Indeterminacy in concurrent computation).

Nevertheless, attempts were made to extend logic programming to concurrent computation. However, Hewitt and Agha [1991] claimed that the resulting systems were not deductive in the following sense: computational steps of the concurrent logic programming systems do not follow deductively from previous steps (see Indeterminacy in concurrent computation). Recently, logic programming has been integrated into the actor model in a way that maintains logical semantics.[21]

Migration

Migration in the actor model is the ability of actors to change locations. *E.g.*, in his dissertation, Aki Yonezawa modeled a post office that customer actors could enter, change locations within while operating, and exit. An actor that can migrate can be modeled by having a location actor that changes when the actor migrates. However the faithfulness of this modeling is controversial and the subject of research.

Security

The security of actors can be protected in the following ways:

- hardwiring in which actors are physically connected
- computer hardware as in Burroughs B5000, Lisp machine, *etc.*
- virtual machines as in Java virtual machine, Common Language Runtime, *etc.*
- operating systems as in capability-based systems
- signing and/or encryption of actors and their addresses

Synthesizing addresses of actors

A delicate point in the actor model is the ability to synthesize the address of an actor. In some cases security can be used to prevent the synthesis of addresses (see Security). However, if an actor address is simply a bit string then clearly it can be synthesized although it may be difficult or even infeasible to guess the address of an actor if the bit strings are long enough. SOAP uses a URL for the address of an endpoint where an actor can be reached. Since a URL is a character string, it can clearly be synthesized although encryption can make it virtually impossible to guess.

Synthesizing the addresses of actors is usually modeled using mapping. The idea is to use an actor system to perform the mapping to the actual actor addresses. For example, on a computer the memory structure of the computer can be modeled as an actor system that does the mapping. In the case of SOAP addresses, it's modeling the DNS and the rest of the URL mapping.

Contrast with other models of message-passing concurrency

Robin Milner's initial published work on concurrency[23] was also notable in that it was not based on composing sequential processes. His work differed from the actor model because it was based on a fixed number of processes of fixed topology communicating numbers and strings using synchronous communication. The original communicating sequential processes (CSP) model[24] published by Tony Hoare differed from the actor model because it was based on the parallel composition of a fixed number of sequential processes connected in a fixed topology, and communicating using synchronous message-passing based on process names (see Actor model and process calculi history). Later versions of CSP abandoned communication based on process names in favor of anonymous communication via channels, an approach also used in Milner's work on the calculus of communicating systems and the π-calculus.

These early models by Milner and Hoare both had the property of bounded nondeterminism. Modern, theoretical CSP ([Hoare 1985] and [Roscoe 2005]) explicitly provides unbounded nondeterminism.

Petri nets and their extensions (e.g., coloured Petri nets) are like actors in that they are based on asynchronous message passing and unbounded nondeterminism, while they are like early CSP in that they define fixed topologies of elementary processing steps (transitions) and message repositories (places).

Influence

The actor model has been influential on both theory development and practical software development.

Theory

The actor model has influenced the development of the π-calculus and subsequent process calculi. In his Turing lecture, Robin Milner wrote:[25]

> Now, the pure lambda-calculus is built with just two kinds of thing: terms and variables. Can we achieve the same economy for a process calculus? Carl Hewitt, with his actors model, responded to this challenge long ago; he declared that a value, an operator on values, and a process should all be the same kind of thing: an actor.
>
> This goal impressed me, because it implies the homogeneity and completeness of expression ... But it was long before I could see how to attain the goal in terms of an algebraic calculus...

So, in the spirit of Hewitt, our first step is to demand that all things denoted by terms or accessed by names—values, registers, operators, processes, objects—are all of the same kind of thing; they should all be processes.

Practice

The actor model has had extensive influence on commercial practice. For example, Twitter has used actors for scalability.[26] Also, Microsoft has used the actor model in the development of its Asynchronous Agents Library.[27] There are many other actor libraries listed in the actor libraries and frameworks section below.

Addressed issues

According to Hewitt [2006], the actor model addresses issues in computer and communications architecture, concurrent programming languages, and Web services including the following:

- Scalability: the challenge of scaling up concurrency both locally and nonlocally.
- Transparency: bridging the chasm between local and nonlocal concurrency. Transparency is currently a controversial issue. Some researchers have advocated a strict separation between local concurrency using concurrent programming languages (e.g., Java and C#) from nonlocal concurrency using SOAP for Web services. Strict separation produces a lack of transparency that causes problems when it is desirable/necessary to change between local and nonlocal access to Web services (see Distributed computing).
- Inconsistency: inconsistency is the norm because all very large knowledge systems about human information system interactions are inconsistent. This inconsistency extends to the documentation and specifications of very large systems (e.g., Microsoft Windows software, etc.), which are internally inconsistent.

Many of the ideas introduced in the actor model are now also finding application in multi-agent systems for these same reasons [Hewitt 2006b 2007b]. The key difference is that agent systems (in most definitions) impose extra constraints upon the actors, typically requiring that they make use of commitments and goals.

Programming with actors

A number of different programming languages employ the actor model or some variation of it. These languages include:

Early actor programming languages

- Act 1, 2 and 3[28][29]
- Acttalk[30]
- Ani[31]
- Cantor[32]
- Rosette[33]

Later actor programming languages

- ABCL
- AmbientTalk[34]

- Axum[35]
- CAL Actor Language
- D
- Dart
- E
- Elixir
- Erlang
- Fantom
- Humus[36]
- Io
- LFE
- Encore[37]
- Pony[38][39]
- Ptolemy Project
- P[40]
- P#[41]
- Rebeca Modeling Language
- Reia
- SALSA[42]
- Scala[43][44]
- TNSDL

Actor libraries and frameworks

Actor libraries or frameworks have also been implemented to permit actor-style programming in languages that don't have actors built-in. Some of these frameworks are:

Name	Status	Latest release	License	Languages
ReActed (https://github.com/reacted-io/reacted)	Active	2021-09-05	Apache 2.0	Java
Acteur (https://github.com/DavidBM/acteur-rs)	Active	2020-04-16[45]	Apache-2.0 / MIT	Rust
Bastion (https://github.com/bastion-rs/bastion)	Active	2020-08-12[46]	Apache-2.0 / MIT	Rust
Actix (https://github.com/actix/actix)	Active	2020-09-11[47]	MIT	Rust
Aojet (https://github.com/aojet/Aojet)	Active	2016-10-17	MIT	Swift
Actor (https://github.com/edescourtis/actor)	Active	2017-03-09	MIT	Java
Actor4j (https://github.com/relvaner/actor4j-core)	Active	2020-01-31	Apache 2.0	Java
Actr (https://github.com/zakgof/actr)	Active	2019-04-09[48]	Apache 2.0	Java
Vert.x (http://vertx.io)	Active	2018-02-13	Apache 2.0	Java, Groovy, Javascript, Ruby, Scala, Kotlin, Ceylon
ActorFx (https://archive.codeplex.com/?p=actorfx)	Inactive	2013-11-13	Apache 2.0	.NET
Akka (toolkit)	Active	2019-05-21[49]	Apache 2.0	Java and Scala
Akka.NET (http://getakka.net)	Active	2020-08-20[50]	Apache 2.0	.NET
Remact.Net (https://github.com/steforster/Remact.Net)	Inactive	2016-06-26	MIT	.NET, Javascript
Ateji PX (https://web.archive.org/web/20100725024213/http://www.ateji.com/px/)	Inactive	?	?	Java
czmq (http://czmq.zeromq.org/manual:zactor)	Active	2016-11-10	MPL-2	C
F# MailboxProcessor	Active	same as F# (built-in core library)	Apache License	F#
Korus (https://code.google.com/p/korus/)	Active	2010-02-04	GPL 3	Java
Kilim (http://kilim.malhar.net/)[51]	Active	2018-11-09[52]	MIT	Java
ActorFoundry (based on Kilim)	Inactive	2008-12-28	?	Java
ActorKit (https://github.com/stevedekorte/ActorKit)	Active	2011-09-13[53]	BSD	Objective-C
Cloud Haskell (https://haskell-distributed.github.com/wiki.html)	Active	2015-06-17[54]	BSD	Haskell
CloudI (http://cloudi.org)	Active	2021-05-	MIT	ATS, C/C++, Elixir/Erlang/LFE,

		27[55]		Go, Haskell, Java, Javascript, OCaml, Perl, PHP, Python, Ruby
Clutter (https://wiki.gnome.org/Projects/Clutter)	Active	2017-05-12[56]	LGPL 2.1	C, C++ (cluttermm), Python (pyclutter), Perl (perl-Clutter)
NAct (https://code.google.com/p/n-act/)	Inactive	2012-02-28	LGPL 3.0	.NET
Nact (https://nact.io/)	Active	2018-06-06[57]	Apache 2.0	JavaScript/ReasonML
Retlang (https://code.google.com/p/retlang/)	Inactive	2011-05-18[58]	New BSD	.NET
JActor (https://web.archive.org/web/20140808051834/http://jactorconsulting.com/product/jactor/)	Inactive	2013-01-22	LGPL	Java
Jetlang (https://code.google.com/p/jetlang/)	Active	2013-05-30[59]	New BSD	Java
Haskell-Actor (https://code.google.com/p/haskellactor/)	Active?	2008	New BSD	Haskell
GPars (http://gpars.org/)	Active	2014-05-09[60]	Apache 2.0	Groovy
OOSMOS (https://www.oosmos.com/)	Active	2019-05-09[61]	GPL 2.0 and commercial (dual licensing)	C. C++ friendly
Panini (http://www.cs.iastate.edu/~panini/)	Active	2014-05-22	MPL 1.1	Programming Language by itself
PARLEY (https://web.archive.org/web/20100616003529/http://osl.cs.uiuc.edu/parley/)	Active?	2007-22-07	GPL 2.1	Python
Peernetic (https://github.com/offbynull/peernetic)	Active	2007-06-29	LGPL 3.0	Java
Picos (http://picolabs.io/)	Active	2020-02-04	MIT	KRL
PostSharp (http://doc.postsharp.net/actor)	Active	2014-09-24	Commercial / Freemium	.NET
Pulsar (https://pypi.org/project/pulsar/)	Active	2016-07-09[62]	New BSD	Python
Pulsar (https://github.com/puniverse/pulsar)	Active	2016-02-18[63]	LGPL/Eclipse	Clojure
Pykka (http://pykka.readthedocs.org/en/latest/index.html)	Active	2019-05-07[64]	Apache 2.0	Python
Termite Scheme (https://code.google.com/p/termite/)	Active?	2009-05-21	LGPL	Scheme (Gambit implementation)
Theron (https://web.archive.org/web/20140810090245/http://www.theron-library.com/)	Inactive[65]	2014-01-18[66]	MIT[67]	C++
Thespian (https://thespianpy.com)	Active	2020-03-10	MIT	Python
Quasar (https://github.com/puniverse/quasar)	Active	2018-11-02[68]	LGPL/Eclipse	Java

Libactor (https://code.google.com/p/libactor/)	Active?	2009	GPL 2.0	C
Actor-CPP (https://code.google.com/p/actor-cpp/)	Active	2012-03-10[69]	GPL 2.0	C++
S4 (http://incubator.apache.org/s4/)	Inactive	2012-07-31[70]	Apache 2.0	Java
C++ Actor Framework (CAF) (http://actor-framework.org/)	Active	2020-02-08[71]	Boost Software License 1.0 and BSD 3-Clause	C++11
Celluloid (https://github.com/celluloid/celluloid/)	Active	2018-12-20[72]	MIT	Ruby
LabVIEW Actor Framework (http://ni.com/actorframework)	Active	2012-03-01[73]	National Instruments SLA (http://www.ni.com/legal/license/)	LabVIEW
LabVIEW Messenger Library (https://lavag.org/files/file/220-messenger-library/)	Active	2021-05-24	BSD	LabVIEW
Orbit (http://www.orbit.cloud)	Active	2019-05-28[74]	New BSD	Java
QP frameworks for real-time embedded systems	Active	2019-05-25[75]	GPL 2.0 and commercial (dual licensing)	C and C++
libprocess (https://github.com/3rdparty/libprocess)	Active	2013-06-19	Apache 2.0	C++
SObjectizer (https://bitbucket.org/sobjectizerteam/sobjectizer/)	Active	2020-05-09[76]	New BSD	C++11
rotor (https://github.com/basiliscos/cpp-rotor)	Active	2020-10-23[77]	MIT License	C++17
Orleans (https://dotnet.github.io/orleans/)	Active	2019-06-02[78]	MIT License	C#/.NET
Skynet (https://github.com/cloudwu/skynet)	Active	2020-12-10	MIT License	C/Lua
Reactors.IO (http://reactors.io/)	Active	2016-06-14	BSD License	Java/Scala
libagents (http://itgroup.ro/libagents)	Active	2020-03-08	Free software license	C++11
Proto.Actor (https://github.com/AsynkronIT)	Active	2021-01-05	Free software license	Go, C#, Python, JavaScript, Java, Kotlin
FunctionalJava (https://www.functionaljava.org/)	Active	2018-08-18[79]	BSD 3-Clause	Java
Riker (https://riker.rs/)	Active	2019-01-04	MIT License	Rust
Comedy (https://github.com/untu/comedy)	Active	2019-03-09	EPL 1.0	JavaScript
vlingo (https://github.com/vlingo/vlingo-actors)	Active	2020-07-26	Mozilla Public License 2.0	Java, Kotlin, soon .NET
wasmCloud (https://github.com/wasmcloud)	Active	2021-03-23	Apache 2.0	WebAssembly (Rust, TinyGo, Zig, AssemblyScript)

ray (https://github.com/ray-project/ray)	Active	2020-08-27	Apache 2.0	Python

See also

- Data flow
- Gordon Pask
- Input/output automaton
- Scientific community metaphor

References

1. Hewitt, Carl; Bishop, Peter; Steiger, Richard (1973). "A Universal Modular Actor Formalism for Artificial Intelligence". IJCAI.
2. William Clinger (June 1981). "Foundations of Actor Semantics". Mathematics Doctoral Dissertation. MIT. hdl:1721.1/6935 (https://hdl.handle.net/1721.1%2F6935).
3. Irene Greif (August 1975). "Semantics of Communicating Parallel Processes". EECS Doctoral Dissertation. MIT.
4. Henry Baker; Carl Hewitt (August 1977). "Laws for Communicating Parallel Processes". IFIP.
5. "Laws for Communicating Parallel Processes" (http://dspace.mit.edu/bitstream/handle/1721.1/41962/AI_WP_134A.pdf) (PDF). 10 May 1977.
6. Gul Agha (1986). "Actors: A Model of Concurrent Computation in Distributed Systems". Doctoral Dissertation. MIT Press. hdl:1721.1/6952 (https://hdl.handle.net/1721.1%2F6952).
7. "Home" (https://web.archive.org/web/20130222175604/http://osl.cs.uiuc.edu/). Osl.cs.uiuc.edu. Archived from the original (http://osl.cs.uiuc.edu) on 2013-02-22. Retrieved 2012-12-02.
8. Carl Hewitt. *Viewing Control Structures as Patterns of Passing Messages (http://www.dtic.mil/get-tr-doc/pdf?AD=ADA038246)* Journal of Artificial Intelligence. June 1977.
9. Gul Agha; Ian Mason; Scott Smith; Carolyn Talcott (January 1993). "A Foundation for Actor Computation". *Journal of Functional Programming*.
10. Carl Hewitt (2006-04-27). "What is Commitment? Physical, Organizational, and Social" (http://www.pcs.usp.br/~coin-aamas06/10_commitment-43_16pages.pdf) (PDF). COIN@AAMAS.
11. Mauro Gaspari; Gianluigi Zavattaro (May 1997). "An Algebra of Actors" (https://link.springer.com/content/pdf/10.1007/978-0-387-35562-7_2.pdf) (PDF). *Formal Methods for Open Object-Based Distributed Systems*. Technical Report UBLCS-97-4. University of Bologna. pp. 3–18. doi:10.1007/978-0-387-35562-7_2 (https://doi.org/10.1007%2F978-0-387-35562-7_2). ISBN 978-1-4757-5266-3.
12. M. Gaspari; G. Zavattaro (1999). "An Algebra of Actors". Formal Methods for Open Object Based Systems.
13. Gul Agha; Prasanna Thati (2004). "An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language" (https://web.archive.org/web/20040420064252/http://formal.cs.uiuc.edu/papers/ATactors_festschrift.pdf) (PDF). From OO to FM (Dahl Festschrift) LNCS 2635. Archived from the original (http://formal.cs.uiuc.edu/papers/ATactors_festschrift.pdf) (PDF) on 2004-04-20.
14. John Darlington; Y. K. Guo (1994). "Formalizing Actors in Linear Logic". International Conference on Object-Oriented Information Systems.
15. "What is the Actor Model & When Should You Use it?" (https://mattferderer.com/). *Matt Ferderer*. Retrieved 2021-08-25.

16. Cheung, Leo (2017-07-25). "Why Akka and the actor model shine for IoT applications" (https://www.infoworld.com/article/3209728/why-akka-and-the-actor-model-shine-for-iot-applications.html). *InfoWorld*. Retrieved 2021-08-25.
17. Hansen, Per Brinch (2002). *The Origins of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Springer. ISBN 978-0-387-95401-1.
18. Hansen, Per Brinch (1996). "Monitors and Concurrent Pascal: A Personal History". *Communications of the ACM*: 121–172.
19. Hoare, Tony (October 1974). "Monitors: An Operating System Structuring Concept". *Communications of the ACM*. **17** (10): 549–557. doi:10.1145/355620.361161 (https://doi.org/10.1145%2F355620.361161). S2CID 1005769 (https://api.semanticscholar.org/CorpusID:1005769).
20. Hansen, Per Brinch (July 1973). *Operating System Principles*. Prentice-Hall.
21. Hewitt, Carl (2012). "What is computation? Actor Model versus Turing's Model". In Zenil, Hector (ed.). *A Computable Universe: Understanding Computation & Exploring Nature as Computation. Dedicated to the memory of Alan M. Turing on the 100th anniversary of his birth*. World Scientific Publishing Company.
22. Frederick Knabe. A Distributed Protocol for Channel-Based Communication with Choice PARLE 1992 (https://www.researchgate.net/profile/Frederick_Knabe/publication/2823488_A_Distributed_Protocol_for_Channel-Based_Communication_with_Choice/links/00463536b853ab72ef000000.pdf).
23. Robin Milner. Processes: A Mathematical Model of Computing Agents in Logic Colloquium 1973.
24. C.A.R. Hoare. Communicating sequential processes (http://portal.acm.org/citation.cfm?id=359585&dl=GUIDE&coll=GUIDE&CFID=19884966&CFTOKEN=55490895) CACM. August 1978.
25. Milner, Robin (1993). "Elements of interaction" (https://doi.org/10.1145%2F151233.151240). *Communications of the ACM*. **36**: 78–89. doi:10.1145/151233.151240 (https://doi.org/10.1145%2F151233.151240).
26. "How Twitter Is Scaling « Waiming Mok's Blog" (https://waimingmok.wordpress.com/2009/06/27/how-twitter-is-scaling/). Waimingmok.wordpress.com. 2009-06-27. Retrieved 2012-12-02.
27. "Actor-Based Programming with the Asynchronous Agents Library (https://msdn.microsoft.com/magazine/623b6c0f-c229-4fcd-8a9d-a5ef24c60db9)" MSDN September 2010.
28. Henry Lieberman (June 1981). "A Preview of Act 1". MIT AI memo 625. hdl:1721.1/6350 (https://hdl.handle.net/1721.1%2F6350).
29. Henry Lieberman (June 1981). "Thinking About Lots of Things at Once without Getting Confused: Parallelism in Act 1". MIT AI memo 626. hdl:1721.1/6351 (https://hdl.handle.net/1721.1%2F6351).
30. Jean-Pierre Briot. Acttalk: A framework for object-oriented concurrent programming-design and experience 2nd France-Japan workshop. 1999. (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.2797)
31. Ken Kahn. A Computational Theory of Animation (https://dspace.mit.edu/bitstream/handle/1721.1/41979/AI_WP_145.pdf?sequence=1) MIT EECS Doctoral Dissertation. August 1979.
32. William Athas and Nanette Boden Cantor: An Actor Programming System for Scientific Computing (http://resolver.caltech.edu/CaltechAUTHORS:20160420-155432546) in Proceedings of the NSF Workshop on Object-Based Concurrent Programming. 1988. Special Issue of SIGPLAN Notices.
33. Darrell Woelk. Developing InfoSleuth Agents Using Rosette: An Actor Based Language (https://web.archive.org/web/20170831083730/https://pdfs.semanticscholar.org/e0f3/874399101a0f3b29ec389b8f92b515e373f8.pdf) Proceedings of the CIKM '95 Workshop on Intelligent Information Agents. 1995.

34. Dedecker J., Van Cutsem T., Mostinckx S., D'Hondt T., De Meuter W. Ambient-oriented Programming in AmbientTalk. In “Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP), Dave Thomas (Ed.), Lecture Notes in Computer Science Vol. 4067, pp. 230-254, Springer-Verlag.”, 2006
35. Darryl K. Taft (2009-04-17). "Microsoft Cooking Up New Parallel Programming Language" (http://www.eweek.com/c/a/Application-Development/Microsoft-Cooking-Up-New-Parallel-Programming-Language-Axum-868670/). Eweek.com. Retrieved 2012-12-02.
36. "Humus" (http://www.dalnefre.com/wp/humus/). Dalnefre.com. Retrieved 2012-12-02.
37. Brandauer, Stephan; et al. (2015). "Parallel objects for multicores: A glimpse at the parallel language encore". *Formal Methods for Multicore Programming*. Springer International Publishing: 1–56.
38. "The Pony Language" (http://www.ponylang.org).
39. Clebsch, Sylvan; Drossopoulou, Sophia; Blessing, Sebastian; McNeil, Andy (2015). "Deny capabilities for safe, fast actors". *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE! 2015*. pp. 1–12. doi:10.1145/2824815.2824816 (https://doi.org/10.1145%2F2824815.2824816). ISBN 9781450339018. S2CID 415745 (https://api.semanticscholar.org/CorpusID:415745). by Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, Andy McNeil
40. "The P Language" (https://github.com/p-org/P). 2019-03-08.
41. "The P# Language" (https://github.com/p-org/PSharp). 2019-03-12.
42. Carlos Varela and Gul Agha (2001). "Programming Dynamically Reconfigurable Open Systems with SALSA". *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*. **36**.
43. Philipp Haller and Martin Odersky (September 2006). "Event-Based Programming without Inversion of Control" (http://lampwww.epfl.ch/~odersky/papers/jmlc06.pdf) (PDF). Proc. JMLC 2006.
44. Philipp Haller and Martin Odersky (January 2007). "Actors that Unify Threads and Events" (https://web.archive.org/web/20110607225711/http://lamp.epfl.ch/~phaller/doc/haller07coord.pdf) (PDF). Technical report LAMP 2007. Archived from the original (http://lamp.epfl.ch/~phaller/doc/haller07coord.pdf) (PDF) on 2011-06-07. Retrieved 2007-12-10.
45. "acteur - 0.9.1· David Bonet · Crates.io" (https://crates.io/crates/acteur/0.9.1). crates.io. Retrieved 2020-04-16.
46. Bulut, Mahmut (2019-12-15). "Bastion on Crates.io" (https://crates.io/crates/bastion). *Crates.io*. Retrieved 2019-12-15.
47. "actix - 0.10.0· Rob Ede · Crates.io" (https://crates.io/crates/actix/0.10.0). crates.io. Retrieved 2021-02-28.
48. "Releases · zakgof/actr · GitHub" (https://github.com/zakgof/actr/releases). Github.com. Retrieved 2019-04-16.
49. "Akka 2.5.23 Released · Akka" (https://akka.io/blog/news/2019/05/21/akka-2.5.23-released). Akka. 2019-05-21. Retrieved 2019-06-03.
50. Akka.NET v1.4.10 Stable Release *GitHub - akkadotnet/akka.net: Port of Akka actors for .NET.* (https://github.com/akkadotnet/akka.net), Akka.NET, 2020-10-01, retrieved 2020-10-01
51. Srinivasan, Sriram; Alan Mycroft (2008). "Kilim: Isolation-Typed Actors for Java" (http://www.malhar.net/sriram/kilim/kilim_ecoop08.pdf) (PDF). *European Conference on Object Oriented Programming ECOOP 2008*. Cyprus. Retrieved 2016-02-25.
52. "Releases · kilim/kilim · GitHub" (https://github.com/kilim/kilim/releases). Github.com. Retrieved 2019-06-03.
53. "Commit History · stevedekorte/ActorKit · GitHub" (https://github.com/stevedekorte/ActorKit/commits/master). Github.com. Retrieved 2016-02-25.
54. "Commit History · haskell-distributed/distributed-process · GitHub" (https://github.com/haskell-distributed/distributed-process/commits/master). Github.com. Retrieved 2012-12-02.

55. "Releases · CloudI/CloudI · GitHub" (https://github.com/CloudI/CloudI/releases). Github.com. Retrieved 2021-06-21.
56. "Tags · GNOME/clutter · GitLab" (https://gitlab.gnome.org/GNOME/clutter/tags). gitlab.gnome.org. Retrieved 2019-06-03.
57. "Releases · ncthbrt/nact · GitHub" (https://github.com/ncthbrt/nact/releases). Retrieved 2019-06-03.
58. "Changes - retlang - Message based concurrency in .NET - Google Project Hosting" (https://code.google.com/p/retlang/source/list). Retrieved 2016-02-25.
59. "jetlang-0.2.9-bin.zip - jetlang - jetlang-0.2.9-bin.zip - Message based concurrency for Java - Google Project Hosting" (https://code.google.com/p/jetlang/downloads/detail?name=jetlang-0.2.9-bin.zip&can=2&q=). 2012-02-14. Retrieved 2016-02-25.
60. "GPars Releases" (https://github.com/GPars/GPars/releases). GitHub. Retrieved 2016-02-25.
61. "Releases · oosmos/oosmos · GitHub" (https://github.com/oosmos/oosmos/releases). GitHub. Retrieved 2019-06-03.
62. "Pulsar Design and Actors" (https://web.archive.org/web/20150704114118/http://pythonhosted.org/pulsar/design.html#actors). Archived from the original (https://pythonhosted.org/pulsar/design.html#actors) on 2015-07-04.
63. "Pulsar documentation" (https://web.archive.org/web/20130726095621/http://puniverse.github.io/pulsar/manual/core.html). Archived from the original (https://puniverse.github.io/pulsar/manual/core.html) on 2013-07-26.
64. "Changes – Pykka 2.0.0 documentation" (https://www.pykka.org/en/latest/changes/#v2-0-0-2019-05-07). pykka.org. Retrieved 2019-06-03.
65. "Theron – Ashton Mason" (http://www.ashtonmason.net/theron/). Retrieved 2018-08-29.
66. "Theron - Version 6.00.02 released" (https://web.archive.org/web/20160316122155/http://www.theron-library.com/index.php?t=news). Theron-library.com. Archived from the original (http://www.theron-library.com/index.php?t=news) on 2016-03-16. Retrieved 2016-02-25.
67. "Theron" (https://web.archive.org/web/20160304000109/http://www.theron-library.com/index.php?t=page&p=license). Theron-library.com. Archived from the original (http://www.theron-library.com/index.php?t=page&p=license) on 2016-03-04. Retrieved 2016-02-25.
68. "Releases · puniverse/quasar · GitHub" (https://github.com/puniverse/quasar/releases). Retrieved 2019-06-03.
69. "Changes - actor-cpp - An implementation of the actor model for C++ - Google Project Hosting" (https://code.google.com/p/actor-cpp/source/list). Retrieved 2012-12-02.
70. "Commit History · s4/s4 · Apache" (https://web.archive.org/web/20160306073515/https://git1-us-west.apache.org/repos/asf?p=incubator-s4.git). apache.org. Archived from the original (https://git1-us-west.apache.org/repos/asf?p=incubator-s4.git) on 2016-03-06. Retrieved 2016-01-16.
71. "Releases · actor-framework/actor-framework · GitHub" (https://github.com/actor-framework/actor-framework/releases). Github.com. Retrieved 2020-03-07.
72. "celluloid | RubyGems.org | your community gem host" (http://rubygems.org/gems/celluloid). RubyGems.org. Retrieved 2019-06-03.
73. "Community: Actor Framework, LV 2011 revision (version 3.0.7)" (https://decibel.ni.com/content/docs/DOC-18308). Decibel.ni.com. 2011-09-23. Retrieved 2016-02-25.
74. "Releases · orbit/orbit · GitHub" (https://github.com/orbit/orbit/releases). GitHub. Retrieved 2019-06-03.
75. "QP Real-Time Embedded Frameworks & Tools - Browse Files at" (https://sourceforge.net/projects/qpc/files/). Sourceforge.net. Retrieved 2019-06-03.
76. "Releases · Stiffstream/sobjectizer · GitHub" (https://github.com/Stiffstream/sobjectizer/releases). GitHub. Retrieved 2019-06-19.

77. "Releases · basiliscos/cpp-rotor· GitHub" (https://github.com/basiliscos/cpp-rotor/releases). GitHub. Retrieved 2020-10-10.
78. "Releases · dotnet/orleans · GitHub" (https://github.com/dotnet/orleans/releases). GitHub. Retrieved 2019-06-03.
79. "FunctionalJava releases" (https://github.com/functionaljava/functionaljava/releases). GitHub. Retrieved 2018-08-23.

Further reading

- Gul Agha. **Actors: A Model of Concurrent Computation in Distributed Systems (https://apps.dtic.mil/dtic/tr/fulltext/u2/a157917.pdf)**. MIT Press 1985.
- Paul Baran. **On Distributed Communications Networks** IEEE Transactions on Communications Systems. March 1964.
- William A. Woods. **Transition network grammars for natural language analysis (http://files.eric.ed.gov/fulltext/ED037733.pdf)** CACM. 1970.
- Carl Hewitt. **Procedural Embedding of Knowledge In Planner (https://www.ijcai.org/Proceedings/71/Papers/014%20A.pdf)** IJCAI 1971.
- G.M. Birtwistle, Ole-Johan Dahl, B. Myhrhaug and Kristen Nygaard. **SIMULA Begin** Auerbach Publishers Inc, 1973.
- Carl Hewitt, *et al.* **Actor Induction and Meta-evaluation (http://www.academia.edu/download/33831594/actor-induction.pdf)** Conference Record of ACM Symposium on Principles of Programming Languages, January 1974.
- Carl Hewitt, *et https://link.springer.com/chapter/10.1007/3-540-06859-7_147al*. ***Behavioral Semantics of Nonrecursive Control Structure (https://link.springer.com/chapter/10.1007/3-540-06859-7_147)*** *Proceedings of Colloque sur la Programmation, April 1974.*
- Irene Greif and Carl Hewitt. **Actor Semantics of PLANNER-73 (https://dspace.mit.edu/bitstream/handle/1721.1/41116/AI_WP_081.pdf?sequence=4&origin=publication_detail)** Conference Record of ACM Symposium on Principles of Programming Languages. January 1975.
- Carl Hewitt. **How to Use What You Know (https://web.archive.org/web/20190307161903/http://pdfs.semanticscholar.org/fc65/4c70dece00b1e4bbb63453c6ff2c81c0893a.pdf)** IJCAI. September, 1975.
- Alan Kay and Adele Goldberg. **Smalltalk-72 Instruction Manual** (http://www.bitsavers.org.nyud.net/pdf/xerox/parc/techReports/Smalltalk-72_Instruction_Manual_Mar76.pdf) Xerox PARC Memo SSL-76-6. May 1976.
- Edsger Dijkstra. **A discipline of programming** Prentice Hall. 1976.
- Carl Hewitt and Henry Baker **Actors and Continuous Functionals (https://web.archive.org/web/20060919015756/http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-194.pdf)** Proceeding of IFIP Working Conference on Formal Description of Programming Concepts. August 1–5, 1977.
- Carl Hewitt and Russ Atkinson. **Synchronization in Actor Systems (http://portal.acm.org/citation.cfm?id=512975&coll=portal&dl=ACM)** Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1977
- Carl Hewitt and Russ Atkinson. **Specification and Proof Techniques for Serializers (https://web.archive.org/web/20170831085117/https://pdfs.semanticscholar.org/bd25/d3a1ed23c79ff09bccf91ce9affd3b399ebe.pdf)** IEEE Journal on Software Engineering. January 1979.
- Ken Kahn. **A Computational Theory of Animation (https://dspace.mit.edu/bitstream/handle/1721.1/41979/AI_WP_145.pdf?sequence=1)** MIT EECS Doctoral Dissertation. August 1979.

- Carl Hewitt, Beppe Attardi, and Henry Lieberman. **Delegation in Message Passing** Proceedings of First International Conference on Distributed Systems Huntsville, AL. October 1979.
- Nissim Francez, C.A.R. Hoare, Daniel Lehmann, and Willem-Paul de Roever. **Semantics of nondetermiism, concurrency, and communication** Journal of Computer and System Sciences. December 1979.
- George Milne and Robin Milner. **Concurrent processes and their syntax** JACM. April 1979.
- Daniel Theriault. **A Primer for the Act-1 Language** MIT AI memo 672. April 1982.
- Daniel Theriault. **Issues in the Design and Implementation of Act 2 (https://apps.dtic.mil/dtic/tr/fulltext/u2/a132326.pdf)** MIT AI technical report 728. June 1983.
- Henry Lieberman. **An Object-Oriented Simulator for the Apiary** Conference of the American Association for Artificial Intelligence, Washington, D. C., August 1983
- Carl Hewitt and Peter de Jong. **Analyzing the Roles of Descriptions and Actions in Open Systems (http://www.dtic.mil/get-tr-doc/pdf?AD=ADA133614)** Proceedings of the National Conference on Artificial Intelligence. August 1983.
- Carl Hewitt and Henry Lieberman. **Design Issues in Parallel Architecture for Artificial Intelligence** MIT AI memo 750. Nov. 1983.
- C.A.R. Hoare. **Communicating Sequential Processes (http://www.usingcsp.com/)** Prentice Hall. 1985.
- Carl Hewitt. **The Challenge of Open Systems** Byte. April 1985. Reprinted in *The foundation of artificial intelligence: a sourcebook* Cambridge University Press. 1990.
- Carl Manning. **Traveler: the actor observatory** ECOOP 1987. Also appears in Lecture Notes in Computer Science, vol. 276.
- William Athas and Charles Seitz **Multicomputers: message-passing concurrent computers (https://ieeexplore.ieee.org/abstract/document/73/)** IEEE Computer August 1988.
- William Athas and Nanette Boden **Cantor: An Actor Programming System for Scientific Computing** in Proceedings of the NSF Workshop on Object-Based Concurrent Programming. 1988. Special Issue of SIGPLAN Notices.
- Jean-Pierre Briot. **From objects to actors: Study of a limited symbiosis in Smalltalk-80 (https://www.researchgate.net/profile/Jean-Pierre_Briot/publication/234812358_From_objects_to_Actors_Study_of_a_limited_symbiosis_in_Smalltalk-80/links/0c96053bd5ac8322b6000000.pdf)** Rapport de Recherche 88-58, RXF-LITP, Paris, France, September 1988
- William Dally and Wills, D. **Universal mechanisms for concurrency (https://link.springer.com/chapter/10.1007/3540512845_30)** PARLE 1989.
- W. Horwat, A. Chien, and W. Dally. **Experience with CST: Programming and Implementation (http://www.dtic.mil/get-tr-doc/pdf?AD=ADA211882)** PLDI. 1989.
- Carl Hewitt. **Towards Open Information Systems Semantics** Proceedings of 10th International Workshop on Distributed Artificial Intelligence. October 23–27, 1990. Bandera, Texas.
- Akinori Yonezawa, Ed. **ABCL: An Object-Oriented Concurrent System** MIT Press. 1990.
- K. Kahn and Vijay A. Saraswat, "Actors as a special case of concurrent constraint (logic) programming (http://doi.acm.org/10.1145/97946.97955)", in SIGPLAN *Notices*, October 1990. Describes Janus.
- Carl Hewitt. **Open Information Systems Semantics** Journal of Artificial Intelligence. January 1991.
- Carl Hewitt and Jeff Inman. **DAI Betwixt and Between: From "Intelligent Agents" to Open Systems Science (https://web.archive.org/web/20170831090048/https://pdfs.semanticscholar.org/7840/bbf6b2fceb014cd3e8eeb2bd81529c7b36b5.pdf)** IEEE Transactions on Systems, Man, and Cybernetics. Nov./Dec. 1991.
- Carl Hewitt and Gul Agha. **Guarded Horn clause languages: are they deductive and Logical?** International Conference on Fifth Generation Computer Systems, Ohmsha 1988. Tokyo. Also in *Artificial Intelligence at MIT*, Vol. 2. MIT Press 1991.

- William Dally, *et al.* **The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms (https://ieeexplore.ieee.org/abstract/document/127581/)** IEEE Micro. April 1992.
- S. Miriyala, G. Agha, and Y.Sami. **Visualizing actor programs using predicate transition nets (http://osl.cs.illinois.edu/media/papers/miriyala-1992-vlc-visualizing_actor_programs_using_predicate_transition_nets.pdf)** Journal of Visual Programming. 1992.
- Carl Hewitt and Carl Manning. **Negotiation Architecture for Large-Scale Crisis Management (https://web.archive.org/web/20170831125720/https://vvvvw.aaai.org/Papers/Workshops/1994/WS-94-04/WS94-04-008.pdf)** AAAI-94 Workshop on Models of Conflict Management in Cooperative Problem Solving. Seattle, WA. Aug. 4, 1994.
- Carl Hewitt and Carl Manning. **Synthetic Infrastructures for Multi-Agency Systems** Proceedings of ICMAS '96. Kyoto, Japan. December 8–13, 1996.
- S. Frolund. **Coordinating Distributed Objects: An Actor-Based Approach for Synchronization** MIT Press. November 1996.
- W. Kim. **ThAL: An Actor System for Efficient and Scalable Concurrent Computing (https://www.researchgate.net/profile/Wooyoung_Kim2/publication/2308617_Thal_An_Actor_System_For_Efficient_And_Scalable_Concurrent_Computing/links/02e7e517614e73041a000000.pdf)** PhD thesis. University of Illinois at Urbana Champaign. 1997.
- Jean-Pierre Briot. **Acttalk: A framework for object-oriented concurrent programming-design and experience** (https://web.archive.org/web/20030427222407/http://www.ifs.uni-linz.ac.at/~ecoop/cd/papers/ec89/ec890109.pdf) 2nd France-Japan workshop. 1999.
- N. Jamali, P. Thati, and G. Agha. **An actor based architecture for customizing and controlling agent ensembles (https://www.researchgate.net/profile/Gul_Agha/publication/3420461_An_Actor-Based_Architecture_for_Customizing_and_Controlling_Agent_Ensembles/links/55aeb2ed08aed9b7dcdda586.pdf)** IEEE Intelligent Systems. 14(2). 1999.
- Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Nielsen, Satish Thatte, Dave Winer. **Simple Object Access Protocol (SOAP) 1.1** W3C Note. May 2000.
- M. Astley, D. Sturman, and G. Agha. **Customizable middleware for modular distributed software (http://osl.cs.illinois.edu/media/papers/astley-2001-cacm-customizable_middleware_for_modular_distributed_software.pdf)** CACM. 44(5) 2001.
- Edward Lee, S. Neuendorffer, and M. Wirthlin. **Actor-oriented design of embedded hardware and software systems** (http://ptolemy.eecs.berkeley.edu/papers/02/actorOrientedDesign/newFinal.pdf) *Journal of Circuits, Systems, and Computers*. 2002.
- P. Thati, R. Ziaei, and G. Agha. **A Theory of May Testing for Actors** Formal Methods for Open Object-based Distributed Systems. March 2002.
- P. Thati, R. Ziaei, and G. Agha. **A theory of may testing for asynchronous calculi with locality and no name matching** Algebraic Methodology and Software Technology. Springer Verlag. September 2002. LNCS 2422.
- Stephen Neuendorffer. **Actor-Oriented Metaprogramming** (http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/ERL-05-1.pdf) PhD Thesis. University of California, Berkeley. December, 2004
- Carl Hewitt (2006a) **The repeated demise of logic programming and why it will be reincarnated (https://web.archive.org/web/20171210124010/https://vvvvw.aaai.org/Papers/Symposia/Spring/2006/SS-06-08/SS06-08-003.pdf)** What Went Wrong and Why: Lessons from AI Research and Applications. Technical Report SS-06-08. AAAI Press. March 2006.

- Carl Hewitt (2006b) *What is Commitment? Physical, Organizational, and Social* (http://www.pcs.usp.br/~coin-aamas06/10_commitment-43_16pages.pdf) COIN@AAMAS. April 27, 2006b.
- Carl Hewitt (2007a) **What is Commitment? Physical, Organizational, and Social (Revised)** Pablo Noriega .et al. editors. LNAI 4386. Springer-Verlag. 2007.
- Carl Hewitt (2007b) **Large-scale Organizational Computing requires Unstratified Paraconsistency and Reflection (https://www.researchgate.net/profile/Bob_Wielinga/publication/221456241_Towards_a_Framework_for_Agent_Coordination_and_Reorganization_AgentCoRe/links/0fcfd508a9cd76ca47000000.pdf#page=105)** COIN@AAMAS'07.
- D. Charousset, T. C. Schmidt, R. Hiesgen and M. Wählisch. *Native actors: a scalable software platform for distributed, heterogeneous environments* (https://dx.doi.org/10.1145/2541329.2541336) in AGERE! '13 Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control.

External links

- Hewitt, Meijer and Szyperski: The Actor Model (everything you wanted to know, but were afraid to ask) (http://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask) Microsoft Channel 9. April 9, 2012.
- Functional Java (http://functionaljava.org/) – a Java library that includes an implementation of concurrent actors with code examples in standard Java and Java 7 BGGA style.
- ActorFoundry (https://web.archive.org/web/20090124154231/http://osl.cs.uiuc.edu/af/) – a Java-based library for actor programming. The familiar Java syntax, an ant build file and a bunch of example make the entry barrier very low.
- ActiveJava (http://tristan.aubrey-jones.com/code/?project=third_year_project&dir=/) – a prototype Java language extension for actor programming.
- Akka (http://akka.io) – actor based library in Scala and Java, from Lightbend Inc..
- GPars (http://gpars.org/) – a concurrency library for Apache Groovy and Java
- Asynchronous Agents Library (http://msdn.microsoft.com/en-us/library/dd492627.aspx) – Microsoft actor library for Visual C++. "The Agents Library is a C++ template library that promotes an actor-based programming model and in-process message passing for coarse-grained dataflow and pipelining tasks. "
- ActorThread in C++11 (https://github.com/lightful/syscpp/) – base template providing the gist of the actor model over naked threads in standard C++11