Actor Model of Computation
for
Scalable Robust Information Systems

One IoT is No IoT[1]

Carl Hewitt

This article is dedicated to Alonzo Church and Dana Scott.

Introduction

The Actor Model is a mathematical theory that treats "*Actors*" as the universal conceptual primitives of digital computation.

Hypothesis:[i] **All physically possible digital computation can be directly modeled and implemented using Actors.**

The model has been used both as a framework for a theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent systems. Actors are direct and efficient:

- Digital computation can be efficiently implemented without loss of processing, communication, or storage efficiency
- Digital computation can be directly modeled without requiring extraneous elements, e.g., channels or registers.

Message passing using types is the foundation of system communication:
The Actor Model is founded on principles of authority and accountability, which provide justifications for sending messages and responding to them.

The advent of massive concurrency using many-core computer architectures and the Internet of Things for Intelligent Applications has galvanized interest in the Actor Model.

When an Actor receives a message, it can concurrently:[2]

- send messages to (unforgeable) addresses of Actors that it has;
- create new Actors; [ii]
- designate how to handle the next message it receives.

---

[i] This hypothesis is an update to [Church 1936] that all physically computable functions can be implemented using the λ-calculus. It is a consequence of the Actor Model that there are some computations that *cannot* be implemented in the λ-calculus.

[ii] with new addresses

The Actor Model can be used as a framework for modeling, understanding, and reasoning about, a wide range of concurrent systems. For example:

- Electronic mail (e-mail) can be modeled as an Actor system. Mail accounts are modeled as Actors and email addresses as Actor addresses.
- Web Services can be modeled with endpoints modeled as Actor addresses.
- Objects with locks (e.g. as in Java and C#) can be modeled as Actors.
- Functional and Logic programs can be implemented using Actors.

Actor technology will see significant application for coordinating all kinds of digital information for individuals, groups, and institutions so their information usefully links together.

Information coordination needs to make use of the following information system principles:

- ***Persistence***. *Information is collected and indexed.*
- ***Concurrency****: Work proceeds interactively and concurrently, overlapping in time.*
- ***Quasi-commutativity****: Information can be used regardless of whether it initiates new work or becomes relevant to ongoing work.*
- ***Sponsorship****: Sponsors provide resources for computation, i.e., processing, storage, and communications.*
- ***Pluralism****: Information is heterogeneous, overlapping and often inconsistent. There is no central arbiter of truth.*
- ***Provenance****: The provenance of information is carefully tracked and recorded.*

The Actor Model is intended to provide a foundation for inconsistency robust information coordination. Inconsistency[i] robustness is information system performance[3] in the face of continual pervasive inconsistencies.[ii] Inconsistency robustness is both an observed phenomenon and a desired feature.

The Actor Model is a mathematical theory of computation that treats "*Actors*" as the universal conceptual primitives of concurrent digital computation [Hewitt, Bishop, and Steiger 1973; Hewitt 1977]. The model has been used both as a framework for a theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent systems.

---

[i] An inference system is *inconsistent* when it is possible to derive both a proposition and its negation.

A *contradiction* is manifest when both a proposition and its negation are asserted even if by different parties, *e.g.*, New York Times said "*Snowden is a whistleblower*.", but NSA said "*Snowden is not a whistleblower.*"

[ii] a shift from the previously dominant paradigms of inconsistency denial and inconsistency elimination, *i.e.*, to sweep inconsistencies under the rug.

Unlike previous models of computation, the Actor Model was inspired by physical laws. It was also influenced by programming languages such as, the λ-calculus[i], Lisp [McCarthy *et. al.* 1962], Simula-67 [Dahl and Nygaard 1967] and Smalltalk-72 [Goldberg and Kay 1976], as well as ideas for Petri Nets [Petri 1962], capabilities systems [Dennis and van Horn 1966] and packet switching [Baran 1964]. The advent of massive concurrency through client-cloud computing and many-core computer architectures has galvanized interest in the Actor Model [Hewitt 2009b].

It is important to distinguish the following:

- modeling arbitrary computational systems using Actors.[ii] It is difficult to find physical computational systems (regardless of how idiosyncratic) that cannot be modeled using Actors.
- securely implementing practical computational applications using Actors remains an active area of research and development.

Decoupling the sender from the communications it sends was a fundamental advance of the Actor Model enabling asynchronous communication and control structures as patterns of passing messages [Hewitt 1977].

An Actor can only communicate with another Actor to which it has an address, which can be implemented in a variety of ways:

- direct physical attachment
- memory or disk addresses
- network addresses
- email addresses

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 reception order.

[i] In general Actor systems can be thousands of times faster than the parallel λ-calculus and pure Logic Programs for many Intelligent Applications.

[ii] An Actor can be implemented directly in hardware.

The Actor Model differs from its predecessors and most current models of computation in that the Actor Model assumes the following:

- Concurrent execution in processing a message.
- The following are *not* required by an Actor: a thread[i], a mailbox[ii], a message queue[iii], its own operating system process[iv], *etc*.
- Message passing has the same overhead as looping and procedure calling.
- Primitive Actors can be implemented in hardware.[v]

The Actor Model can be used as a framework for modeling, understanding, and reasoning about, a wide range of concurrent systems.

For example:

- Electronic mail (e-mail) can be modeled as an Actor system. Mail accounts are modeled as Actors and email addresses as Actor addresses.
- Web Services can be modeled with SOAP endpoints modeled as Actor addresses.
- Objects with locks (*e.g.* as in Java and C#) can be modeled as Actors.

Direct communication and asynchrony

The Actor Model is based on one-way asynchronous communication. Once a message has been sent, it is the responsibility of the receiver.[4]

Messages in the Actor Model are decoupled from the sender and are delivered by the system on a best efforts basis.[5] This was a sharp break with previous approaches to models of concurrent computation in which message sending is tightly coupled with the sender and sending a message synchronously transfers it someplace, *e.g.*, to a buffer, queue, mailbox, channel, broker, server, *etc.* or to the "*ether*" or "*environment*" where it temporarily resides. The lack of synchronicity caused a great deal of misunderstanding at the time of the development of the Actor Model and is still a controversial issue.

---

[i] If an Actor was required to have a thread, it would not have internal parallelism.

[ii] If an Actor was required to have a mailbox then, the mailbox would be an Actor that is required to have its own mailbox…

[iii] If an Actor was required to have message queue accessible by the Actor then it would be impossible to dynamically prioritize messages or to remove messages that have timed out.

[iv] If each Actor were required to have its own operating system process, then message communication between Actors on the same computer would be slow because of marshalling.

[v] In some cases, this involves (clocked) one-way messages so message guarantees and exception processing can be different from typical application Actors.

Because message passing is taken as fundamental in the Actor Model, there cannot be any required overhead, *e.g.*, any requirement to use buffers, pipes, queues, classes, channels, *etc*. Prior to the Actor Model, concurrency was defined in low level machine terms.

It certainly is the case that implementations of the Actor Model typically make use of these hardware capabilities. However, there is no reason that the model could not be implemented directly in hardware without exposing any hardware threads, locks, queues, cores, channels, tasks, *etc.* Also, there is no necessary relationship between the number of Actors and the number threads, cores, locks, tasks, queues, *etc*. that might be in use. Implementations of the Actor Model are free to make use of threads, locks, tasks, queues, coherent memory, transactional memory, cores, *etc*. in any way that is compatible with the laws for Actors [Baker and Hewitt 1977].

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 sequential execution is a special case in which the activation order is linear. Also, the Actor Model is based on communication rather that a global state space as in Turing Machines, CSP [Hoare 1978], Java [Sun 1995, 2004], C++11 [ISO 2011], X86 [AMD 2011], *etc.* The Actor Model does *not* take classical sequential processes as primitive and is *not* built on communicating sequential processes.

A natural development of the Actor Model was to allow Actor addresses in messages. A computation might need to send a message to a recipient from which it would later receive a response. The way to do this is to send a communication which has the message along with the address of another Actor called the *customer* along with the message. The recipient could then cause a response message to be sent to the customer.

Indeterminacy and Quasi-commutativity

The Actor Model supports indeterminacy because the reception order of messages can affect future behavior.

Operations are said to be quasi-commutative to the extent that it doesn’t matter in which order they occur. To the extent possible, quasi-commutativity is used to reduce indeterminacy.

Locality and Security

Locality and security are important characteristics of the Actor Model[Baker and Hewitt 1977].[6]

Locality and security mean that in processing a message: an Actor can send messages only to addresses for which it has information by the following means:

1. that it receives in the message
2. that it already had before it received the message
3. that it creates while processing the message.

In the Actor Model, there is no hypothesis of 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.

That there be no single point of failure can be an important aspect of security.

The security of Actor systems can be protected in the following ways:

Strong personal authentication, *e.g.*, using (3D) continuous interactive bio-authentication instead of passwords

Strong, ubiquitous public key authentication so that it can be verified to whom a public key corresponds. Often this authentication can be performed by local bank offices, *etc.* that publish online multi-national directories of public keys in a network of mistrust. Individual citizens can have their own directories of public keys that are used to automatically and invisibly securely communicate with others.

Many citizens will have more than one authenticated public key, which can be authenticated with various levels of security.

Public keys for IoT ownership so that an IoT device has both:

- a public key of its owner, which is installed when ownership is transferred
- its own unique public/private key pair, which is created internally when acquired by the first owner.

An owner can communicate securely with a device by encrypting information using the device's public key. (For efficiency reasons, most communication will actually be performed using symmetric keys encrypted/signed by public keys.) A device takes instructions only from its owner and is allowed to communicate with the external world only through the information coordination system of its owner. The nonprofit Standard IoT Foundation is working to develop standards based on the Actor Model of computation that provide for interoperation among existing and emerging consortium and proprietary corporate IoT standards.

Hardware architecture security
To help cope with the complexity of software systems that can never be made highly secure without hardware assistance including the following:

- *RAM-processor package encryption* (*i.e.* all traffic between a processor package and RAM is encrypted using a uniquely generated key when a package is powered up and which is invisible to all software) to protect an app (*i.e.* a user application, which is technically a process) from the following:
  - operating systems and hypervisors
  - other apps
  - other equipment, e.g., baseband processors, disk controllers, and USB controllers.
- *Hardware Actors* that communicate only using message passing to protect security registers
- *Every-word-tagged architecture* to protect an Actor in an app from other Actors by using a tag on each word of memory that controls how the memory can be used. Each Actor is protected from reading and/or writing by other Actors in its process. Actors can interact only by sending a message to the unforgeable address of another Actor. Existing software (e.g., operating systems, team integrated development environments, mail systems) will need to be upgraded to use tags.

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 in practice using the following:

- every-word-tagged memory
- signing and encryption of messages

Islets

Islets is a system for citizens to coordinate with IoT devices and other citizens while protecting their sensitive information[i] and fostering (international) commerce.[ii]

Sensitive information can be stored locally in Islets on users' own equipment encrypted with user keys and can be backed up elsewhere encrypted using users' keys. Furthermore, users can share Islet information that they select with other parties -- encrypted with the public keys of other parties so that it be read only by the intended party.

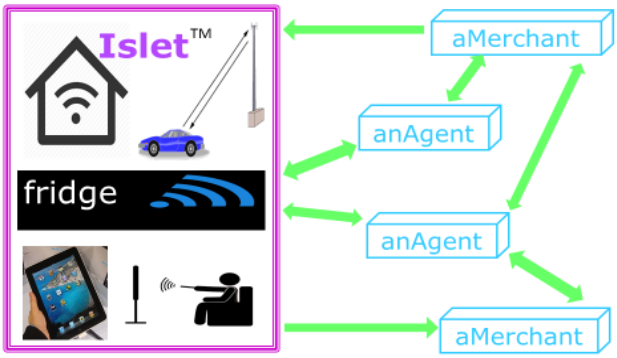
Islets can improve information coordination over current systems that cannot coordinate among numerous competing services (such as Facebook and Google) and numerous fiercely competing merchants (such as Amazon, Home Depot, and Walmart

[i] including sensitive health, legal, political, and social information

[ii] by enabling conclusive verification that foreign governments are no longer conducting mass surveillance using IoT manufactured products and Internet company service provider datacenters.[11][11*][25][36]

using multiple IoT devices from competing manufacturers (such as LG, Nest, Samsung, and Whirlpool). Islet-facilitated coordination can include integration of commerce (such as home, retail, food, travel, and auto), wellness (such as recreation, biometrics, nutrition, exercise, spirituality, medical, and learning), finance (such as banking, investments, and taxes), IoT (such as food management, security, energy management, infotainment, transportation, and communication), social (such as schedule, friends, and family), and work (such as contacts, schedule, and colleagues). Islets can provide lower communications cost than current systems because it is not necessary for users and their IoT devices to always communicate with datacenters.[Burnside, et. al. 2002; Lee and Neuendorffer 2015] Islets also can provide faster response and more robustness because local operations can be faster and more reliable than being required to always use communication links with potentially-overloaded remote datacenters.

Islets need a convenient, effective, high-profitable business model, which must be more effective and efficient than the current datacenters system based on consumer surveillance to improve advertising targeting. Instead, an Islet running on a consumer's equipment can seek out and help evaluate appropriate offers from commerce agents. Such commerce agents can earn commissions and fees from merchants when the referral is exercised. Consequently, merchants will no longer be burdened by having to pay for *grossly inefficient* advertising that annoys potential customers. Instead, businesses can provide their information to commerce agents that aggregate and package it for users' Islets to be used in evaluating offers that can be filtered and ranked according to citizen needs and preferences. All of the convenience currently available through individual company access points must be improved in effectiveness and response time including scalable search and operations that can query commercial datacenters (such as Amazon, Facebook, Google, and LinkedIn) as well as other Islets.



Islet Coordinating with Agents and Merchants Business Model

Outside of citizens' Islets information protected against self-incrimination, governments (through subpoena) will be able to obtain sufficient information for law enforcement including financial transactions, physical movements outside the home, and cell tower tracking information.

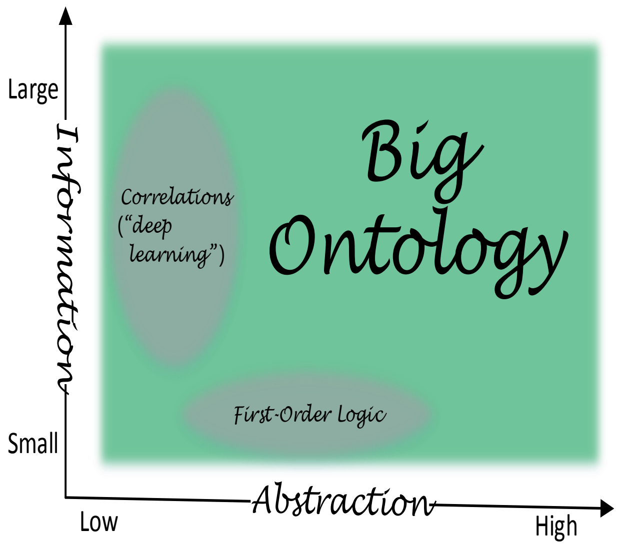
Islets have large amounts of pervasively inconsistent information because IoT devices are only intermittently connected resulting in delayed coordination and because their sources of information are inconsistent including human input and information from the Web.

Unfortunately, current computer information systems lack fundamental inference capabilities needed by Islets. The two most common approaches, formalization using Classical First Order Logic, and statistical reasoning using machine learning, can fail catastrophically in the face of inconsistent information.

- In classical logic, any possible conclusion logically follows from a (hidden) inconsistency.
- In current “Deep Learning” correlation engines inconsistencies are treated as “noise” that can produce unstable probability assessments washing out the ability to draw reliable conclusions.

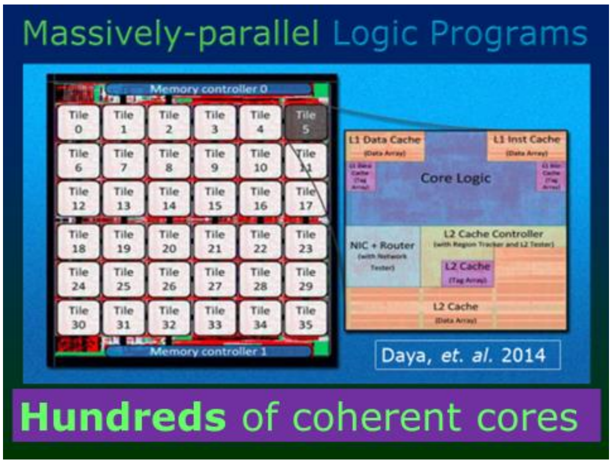
Big Ontology Semantic Systems: Future of Big Info and Analytics

Processing massive amounts of information is a hallmark of the next generation of Intelligent Applications in which pervasively inconsistent information must be coherently understood. Inconsistency Robustness[Hewitt and Woods assisted by Spurr 2015; Meyer 2015] is the science and engineering of large systems with continual, pervasive inconsistencies, a shift from the previously dominant paradigms of inconsistency denial and elimination. Inconsistency robust logic is an important conceptual advance that requires that nothing “extra” can be inferred just from the presence of a contradiction. It improves on both the classical logic and machine learning responses to inconsistency, allowing computer systems to employ the kinds of common sense reasoning strategies used by people.



Direct Logic is a framework in which propositions have arguments for and against. Inference rules provide arguments provide justification for the inference of further propositions. Direct Logic is a bookkeeping system that helps keep track. It doesn’t tell what to do when an inconsistency is derived. But it does have the great virtue that it doesn’t make the mistakes of 1st order logic when reasoning about inconsistent information.

Big Ontology systems need inconsistency robust reasoning[Hewitt and Woods assisted by Spurr 2015; Meyer 2015; Hewitt "Big Ontology" 2017] because they have numerous implicit unknown inconsistencies as well as numerous known ones:



- Inconsistencies in a Big Ontology System are ineradicable and can be subtle.
- There is no practical way to test for inconsistency in a Big Ontology System.
- Big Ontology information can be meaningful, coherent, and useful [Law 2004] although pervasively inconsistent.

Inconsistency Robustness needs massive closely-coupled concurrency for its implementation for which the "Actor" abstraction is ideally suited[Hewitt "Actor Model" 2017, "Logic Programs" 2017,] both as a framework for modeling, and as the basis for practical implementations[Bonér 2016; Bernstein, Bykov, Geller, Kliot, and Thelin 2014]. Actors provide modularity and security without imposing any overhead on computation and storage. Existing IoT legacy systems can be extended using Actors without requiring their re-implementation.

Inconsistency Robustness reasoning facilitates systems development by removing the requirement that a system must attempt to maintain absolute consistency of its information (as in a relational database). Removing this requirement facilitates massive parallelism in reasoning using coherent many-core computers [Daya, et. al. 2014; Jones 2017].

Big Ontology systems gain performance through the following:

- massive parallelism in forward/backward, descriptive, and statistical inference
- synergy by bringing together processing on assertions, goals, and descriptions

Propositional Big Ontology Information

The Art of War can be used to illustrate Logic Programs. For example, “Appear strong when weak and appear weak when strong.” might be applicable when there is a goal to bluff as expressed in the following Logic Programs:

When goal BluffOpponents[x:**Army**] ∴ (**When goal** Weak[x] ∴ SetGoal Appear[Strong[x]])[i]

When goal BluffOpponents[x:**Army**] ∴ (**When assertion** Strong[x] ∴ SetGoal Appear[Weak[x]])[ii]

To facilitate inconsistency robustness, Logic Programs can provide both fine-grained control over both forward and backward inference as illustrated below:

- Forward inference
  - **When assertion** Strong[x:**Army**] ∴ Assert **not** Weak[x][iii]
  - **When assertion** Weak[x:**Army**] ∴ Assert **not** Strong[x][iv]
- Backward inference
  - **When goal** Weak[x:**Army**] ∴ SetGoal **not** Strong[x][v]
  - **When goal** Strong[x:**Army**] ∴ SetGoal **not** Weak[x][vi]

---

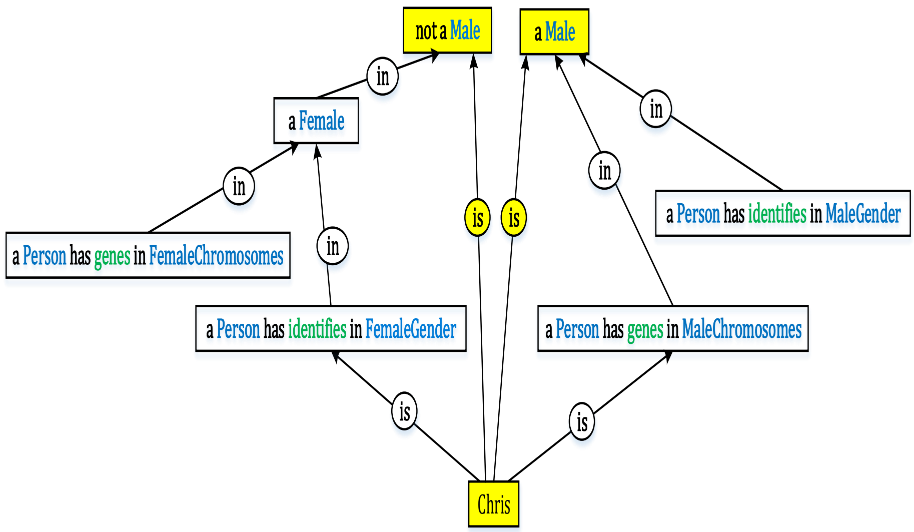
[i] When goal to bluff opponents, then when weak, set goal to appear strong.
[ii] When goal to bluff opponents, then when strong, set goal to appear weak.
[iii] When asserted that x is strong, assert that x is not weak.
[iv] When asserted that x is weak, assert that x is not strong.
[v] When goal that x is a strong, set subgoal that x is not weak.
[vi] When goal that x is a weak, set subgoal that x is not strong.

Logic Programs must be robust against inconsistencies. For example, the common situation of having multiple simultaneous goals (e.g. to both bluff opponents and to intimidate them) often leads to conflict, e.g., adding the following logic program to the ones above:

When goal IntimidateOpponents[x:Army] ∴ SetGoal Appear[Strong[x]][i]

Descriptive Big Ontology Information

Inconsistencies commonly arise in large ontologies. For example, the following can arise in the case of a transgender person in which disentangling maleness can be very problematical in medical/social contexts.[7]

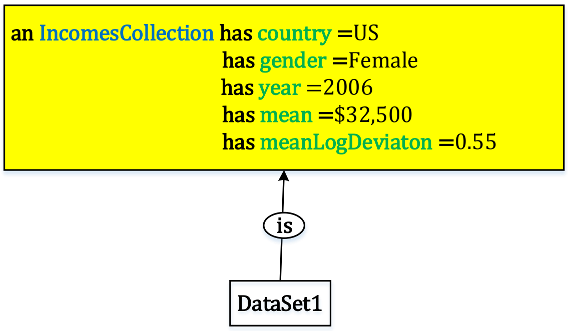


Chris is both a Male and not a Male depending on circumstance

[i] When goal to intimidate the enemy, set subgoal to appear strong.

Statistical Big Ontology Information
Statistical information can be very important in large ontologies.

For example, the diagram at the right depicts DataSet1 in which the 2006 incomes of females in the US had a mean of $32,500 with a mean logarithmic deviation of 0.55.



Statistical Info in Big Ontology System

In summary, Big Ontology Systems make effective use of the following:

- Massive concurrency of hundreds (and then thousands) of coherent-memory many-core processors
- Inconsistency robustness for processing massive amounts of inconsistent information using propositional, descriptive, and statistical inference
- Multiple way to organize, view, and interact with information

3. Resilience

Actors employ message passing using strong types as a fundamental unit of communication because of semantic integrity that is important in IoT operations. Using messages as the architectural unit of communication is a fundamental advance over existing IoT Internet protocols. Existing IoT communication systems are architected on the basis of packets which can be organized into byte streams. However, neither packets nor byte streams are themselves semantically meaningful. Consequently, additional layers of software in current systems must be layered (and then parsed) on top of byte streams to create meaningful messages. Instead, messages should be foundational to network communication and sent encrypted.

Robustness in Runtime Failures

Runtime failures are always a possibility in Actor systems and are dealt with by runtime infrastructures. Message acknowledgement, reception, and response[i] cannot be guaranteed although best efforts are made. Consequences are cleaned up on a best-effort basis.

Robustness in runtime failures is partially based on the following principle:[8]

---

[i] a response is either a returned value or a thrown exception

If an Actor is sent a request, then the continuation *must* be one of the following two mutually exclusive possibilities:

1. to process the response[i] resulting from the recipient receiving the request
2. to throw a **Messaging**[ii] exception[iii]

Just sitting there forever after a request has been sent is a silent failure, which is unacceptable. So, in due course, the infrastructure must throw a **Messaging** exception as governed by the policies in place[iv] if a response (return value or exception) to the request has not been received.

Ideally, if the continuation of sending a request is to throw a **Messaging** exception, then the sender of a response to the request also receives a **Messaging** exception saying that the response could not be processed.

If desired, things can be arranged so that **Messaging** exceptions are distinguished from all other exceptions.

Scalability and Modularity

ActorScript™ is a general purpose programming language for implementing iAdaptive™ concurrency that manages resources and demand. It is differentiated from previous languages by the following:

- Universality
    - Ability to directly specify what Actors can do
    - Specify interface between hardware and software
    - Everything in the language is accomplished using message passing including the very definition of ActorScript itself.
    - Functional, Imperative, Logic, and Concurrent programming are integrated. Concurrency can be dynamically adapted to resources available and current load.
    - Programs do not expose low-level implementation mechanisms such as threads, tasks, channels, coherent memory, location transparency, throttling, load balancing, locks, cores, *etc*. Messages can be directly communicated without requiring indirection through brokers, channels, class hierarchies, mailboxes, pipes, ports, queues *etc*. Variable races are eliminated.
    - Binary XML and JSON are data types.

---

[i] conceptually processed by a customer Actor sent in the request

[ii] A **Messaging** exception can have information concerning the lack of response

[iii] even though the recipient may have received the request and sent a response that has not yet been received by the customer of the request. Requestors need to be able to interact with infrastructures concerning policies to be applied concerning when to generate **Messaging** exceptions.

[iv] For example, several standard deviations have passed in the expected time to receive a response.

    - Application binary interfaces are afforded so that no program symbol need be looked up at runtime.
- Safety and security
    - Programs are extension invariant, i.e., extending a program does not change its meaning.
    - Applications cannot directly harm each other.
- Performance
    - Impose no overhead on implementation of Actor systems
    - Message passing has essentially same overhead as procedure calling and looping.
    - Execution dynamically adjusted for system load and capacity (*e.g.* cores)
    - Locality because execution is not bound by a sequential global memory model
    - Inherent concurrency because execution is not bound by communicating sequential processes
    - Minimize latency along critical paths

ActorScript attempts to achieve the highest level of performance, scalability, and expressibility with a minimum of conceptual primitives.

Scalable information Coordination

Technology now at hand can coordinate all kinds of digital information for individuals, groups, and institutions so their information usefully links together. This coordination can include calendars and to-do lists, communications (including email, SMS, Twitter, Facebook), presence information (including who else is in the neighborhood), physical (including GPS recordings), psychological (including facial expression, heart rate, voice stress) and social (including family, friends, team mates, and colleagues), maps (including firms, points of interest, traffic, parking, and weather), events (including alerts and status), documents (including presentations, spreadsheets, proposals, job applications, health records, photos, videos, gift lists, memos, purchasing, contracts, articles), contacts (including social graphs and reputation), purchasing information (including store purchases, web purchases, GPS and phone records, and buying and travel habits), government information (including licenses, taxes, and rulings), and search results (including rankings and ratings).

Connections

Information coordination works by making connections including examples like the following:

- A statistical connection between "being in a traffic jam" and "driving in downtown Trenton between 5PM and 6PM on a weekday."
- A terminological connection between "MSR" and "Microsoft Research."

- A causal connection between “joining a group” and “being a member of the group.”
- A syntactic connection between “a pin dropped” and “a dropped pin.”
- A biological connection between “a dolphin” and “a mammal”.
- A demographic connection between “undocumented residents of California” and “7% of the population of California.”
- A geographical connection between “Leeds” and “England.”
- A temporal connection between “turning on a computer” and “joining an on-line discussion.”

By making these connections Describer™ information coordination offers tremendous value for individuals, families, groups, and institutions in making more effective use of information technology.

Information Coordination and Action Principles

In practice, coordinated information is invariably inconsistent.[9] Therefore Describer™ must be able to make connections even in the face of inconsistency.[10] The business of Describer is not to make difficult decisions like deciding the ultimate truth or probability of propositions. Instead it provides means for processing information and carefully recording its provenance including arguments (including arguments about arguments) for and against propositions.

Information coordination needs to make use of the following principles:

- ***Persistence**. Information is collected and indexed and no original information is lost.*
- ***Concurrency**: Work proceeds interactively and concurrently, overlapping in time.*
- ***Quasi-commutativity**: Information can be used regardless of whether it initiates new work or becomes relevant to ongoing work.*
- ***Sponsorship**: Sponsors provide resources for computation, i.e., processing, storage, and communications.*
- ***Pluralism**: Information is heterogeneous, overlapping and often inconsistent. There is no central arbiter of truth*
- ***Provenance**: The provenance of information is carefully tracked and recorded*

Interaction creates Reality[11]

> *a philosophical shift in which knowledge is no longer treated primarily as referential, as a set of statements* ***about*** *reality, but as a practice that interferes with other practices. It therefore participates* ***in*** *reality.*
> Annemarie Mol [2002]

Relational physics takes the following view [Laudisa and Rovelli 2008]:[i]

- Relational physics discards the notions of absolute state of a system and absolute properties and values of its physical quantities.
- State and physical quantities refer always to the interaction, or the relation, among multiple systems.
- Nevertheless, relational physics is a complete description of reality.

According to this view, **Interaction creates reality.** Information systems participate in this reality and thus are both consequence and cause.

Tutes[TM]

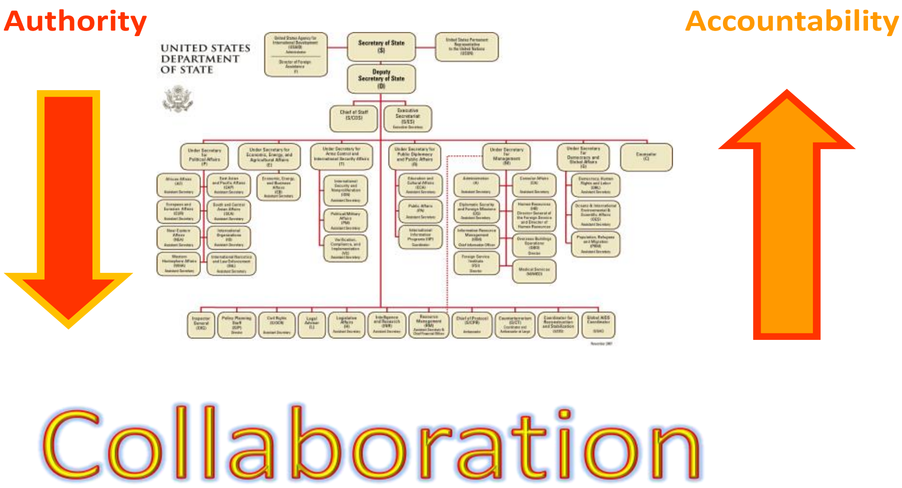
The Actor Model supports authority and accountability in Tutes[ii] with the goal of becoming an effective readily understood approach for addressing scalability issues in Software Engineering. The paradigm takes its inspiration from human institutions. Tutes provide a framework for addressing issues of hierarchy, authority, accountability, scalability, and robustness using methods that are analogous to human institutions. Because humans are very familiar with the principles, methods, and practices of human institutions, they can transfer this knowledge and experience. Tutes achieve scalability using methods and principles similar to those used in human institutions. For example a Tute can have sub-institutions specialized by areas such as sales, production, and so forth. Authority is delegated downward and when necessary issues are escalated upward. Authority requires accountability for its use including record keeping and periodic reports.

---

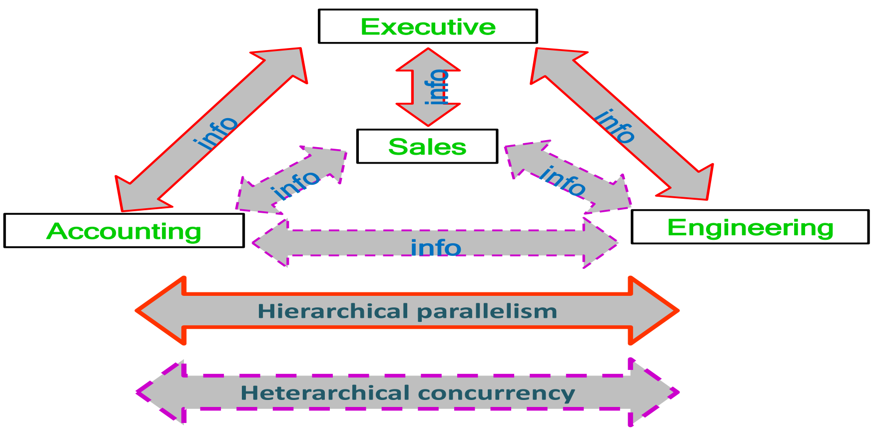
[i] According to [Rovelli 1996]: *Quantum mechanics is a theory about the physical description of physical systems relative to other systems, and this is a complete description of the world.*

[Feynman 1965] offered the following advice: *Do not keep saying to yourself, if you can possibly avoid it, "But how can it be like that?" because you will go "down the drain," into a blind alley from which nobody has yet escaped.*

[ii] Tute[TM] is short for Institution.

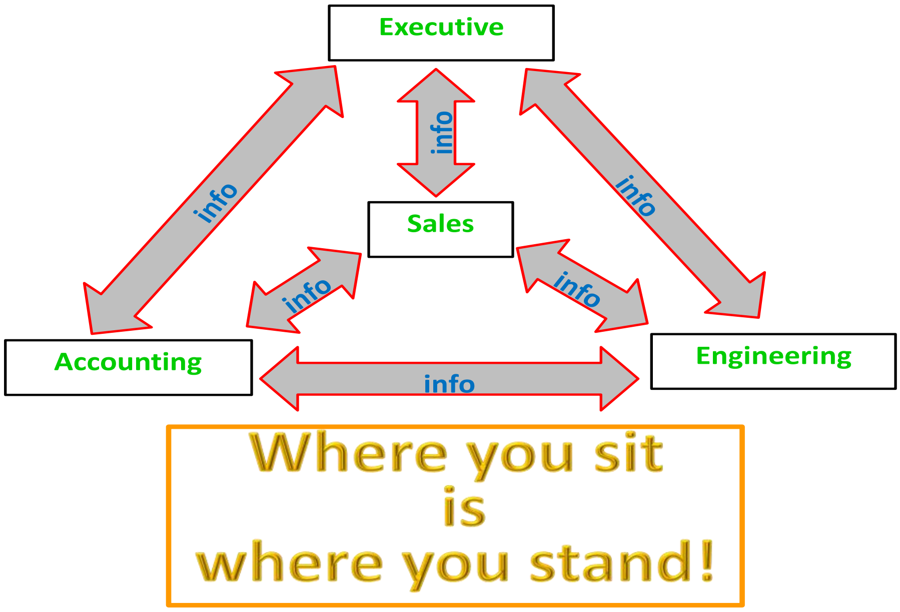


A Tute uses *institutional commitment* defined as information pledged constituting an alliance to go forward. For example, a Tute can use contracts to formalize mutual commitments to fulfill specified obligations.



Scalability of Tutes

Yet, manifestations of information pledged will often be inconsistent. Any given agreement might be internally inconsistent, or two agreements in force at one time could contradict each other.



Inconsistency by Design for Tutes

Issues that arise from such inconsistencies can be negotiated among Tutes. For example the Sales department might have a different view than the Accounting department as to when a transaction should be booked.

Actor Addresses and Implementations

Actor addresses have types. For example the type **Account** has the following interface description:

Account

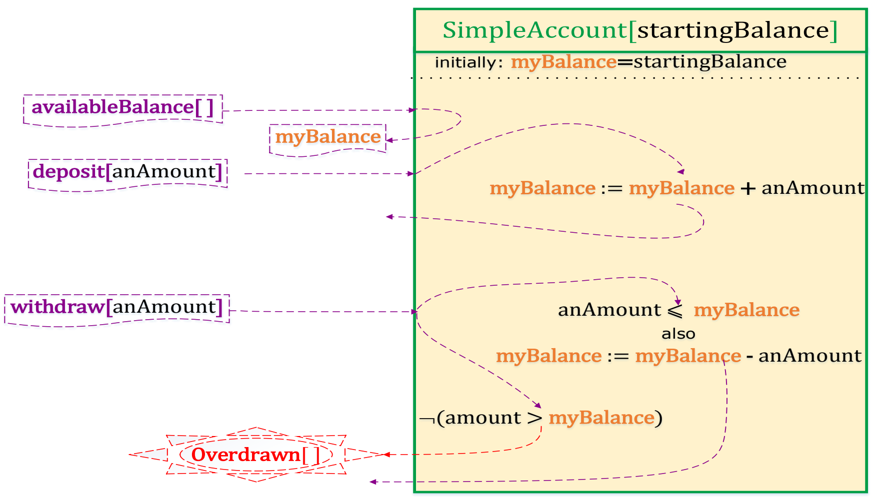
availableBalance[ ]↦Euro

deposit[Euro]↦Void

withdraw[Euro]↦Void

Message Passing

The Actor Model is much more powerful than nondeterministic Turing Machine and λ-calculus models of computation. For example, the Actor for a SimpleAccount diagrammed below cannot be implemented using a nondeterministic Turing Machine or λ-expression:



What a Turing Machine, the λ-calculus, and Pure Logic Programs ***can't*** do

The above diagram is for the implementation of an Actor of type **SimpleAccount** with variable **myBalance**, which behaves as follows:

- If an **availableBalance[ ]** message is received, then return **myBalance**.
- If a **deposit[**anAmount**]** message is received, then increment **myBalance** by anAmount and return.
- If a **withdraw[**anAmount**]** message is received, then if anAmount⩽**myBalance** then decrement **myBalance** by anAmount and return, else throw an **Overdraft[ ]** exception.

Computational Representation Theorem

The *Computational Representation Theorem* [Clinger 1981; Hewitt 2006][12] characterizes computation for systems which are closed in the sense that they do not receive communications from outside:

> The denotation $\mathsf{Denote}_{\mathbf{S}}$ of a closed system $\mathbf{S}$ represents all the possible behaviors of $\mathbf{S}$ as
>
> $$\mathsf{Denote}_{\mathbf{S}} = \lim_{i\to\infty} \mathrm{Progression}_{\mathbf{S}}{}^{i}$$
>
> *where* $\mathrm{Progression}_{\mathbf{S}}$ takes a set of partial behaviors to their next stage, i.e., $\mathrm{Progression}_{\mathbf{S}}{}^{\mathbf{i}} \twoheadrightarrow^{i} \mathrm{Progression}_{\mathbf{S}}{}^{\mathbf{i+1}}$
>
> In this way, $\mathbf{S}$ can be mathematically characterized in terms of all its possible behaviors (including those involving unbounded nondeterminism).[ii]
>
> The denotations form the basis of constructively checking programs against all their possible executions,[iii]

A consequence of the Computational Representation Theorem is that there are uncountably many different Actors.

For example, Real.[ ] can output any real number between 0 and 1 where

Real.[ ] ≡ [(0 **either** 1), ⩛**Postpone** Real.[ ]]

such that

- (0 **either** 1) is the nondeterministic choice of 0 or 1
- [first, ⩛rest] is the list that begins with first and whose remainder is rest
- **Postpone** expression delays execution of expression until the value is needed.

The upshot is that ***concurrent systems can be axiomatized using mathematical logic***[iv] ***but in general cannot be implemented***. Thus, the following practical problem arose:

> How can practical programming languages be rigorously defined since the proposal [Scott and Strachey 1971, Milne and Strachey 1976] to define them in terms λ-calculus failed because the λ-calculus cannot implement concurrency?[13]

A proposed answer to this question is the semantics of ActorScript [Hewitt 2010].

---

[i] read as "*can evolve to*"

[ii] There are no messages in transit in $\mathsf{Denote}_{\mathbf{S}}$

[iii] a restricted form of Model Checking in which the properties checked are limited to those that can be expressed in Linear-time Temporal Logic has been studied [Clarke, Emerson, and Sifakis. ACM 2007 Turing Award]

[iv] including the λ-calculus

Using Implementations versus Interface Extension

Programming languages like ActorScript [Hewitt 2010] take the approach of extending behavior in contrast to the approach of specializing behavior:

- Using implementations: An implement type can make use of other implementations. However, an implementation cannot be subtyped because it is branded to guarantee its behavior that might be violated by subtypes. Consequently, Actors automatically vacuously have the substitution property [Liskov 1987, Liskov and Wing 2001] for implementations.
- Interface extension: A type interface can be extended to have additional message signatures from the type interface that it extends. In general, a system cannot guarantee properties of implementations of an interface type. Consequently, the substitution property may not hold even for Actors that implement the *same* interface.

Language constructs versus Library APIs

Library Application Programming Interfaces (APIs) are an alternative way to introduce concurrency.
For example,

- A limited version of futures[Baker and Hewitt 1977] have been introduced in C++11 [ISO 2011].
- Message Passing Interface (MPI) [Gropp et. al. 1998] provides some ability to pass messages.
- Grand Central Divide provides for queuing tasks.

There are a number of library APIs for Actor-like systems.

In general, appropriately defined language constructs provide greater power, flexibility, and performance than library APIs.[14]

Reasoning about Actor Systems

The principle of Actor induction is:

1. Suppose that an Actor x has property P when it is created
2. Further suppose that if x has property P when it receives a message, then it has property P when it receives the next message.
3. Then x always has the property P.

In his doctoral dissertation, Aki Yonezawa developed further techniques for proving properties of Actor systems including those that make use of migration. Russ Atkinson developed techniques for proving properties of Actors that are guardians of shared resources. Gerry Barber's 1981 doctoral dissertation concerned reasoning about change in knowledgeable office systems.

Other models of concurrency

The Actor Model does not have the following restrictions of other models of concurrency:[15]

- *Single threadedness*: There are no restrictions on the use of threads in implementations.
- *Message delivery order*: There are no restrictions on message delivery order.
- *Independence of sender*: The semantics of a message in the Actor Model is independent of the sender.
- *Lack of garbage collection (automated storage reclamation):* The Actor Model can be used in the following systems:
    - CLR and extensions (Microsoft and Xamarin)
    - JVM (Oracle and IBM)
    - LLVM (Apple)
    - Dalvik (Google)

  In due course, we will need to extend the above systems with a tagged extension of the X86, ARM, and RISC architectures. Many-core architecture has made a tagged extension necessary in order to provide the following:
    - concurrent, nonstop, no-pause automated storage reclamation (garbage collection) and relocation to improve performance,
    - prevention of memory corruption that otherwise results from programming languages like C and C++ using thousands of threads in a process,
    - nonstop migration of Actors (while they are in operation) within a computer and between distributed computers.

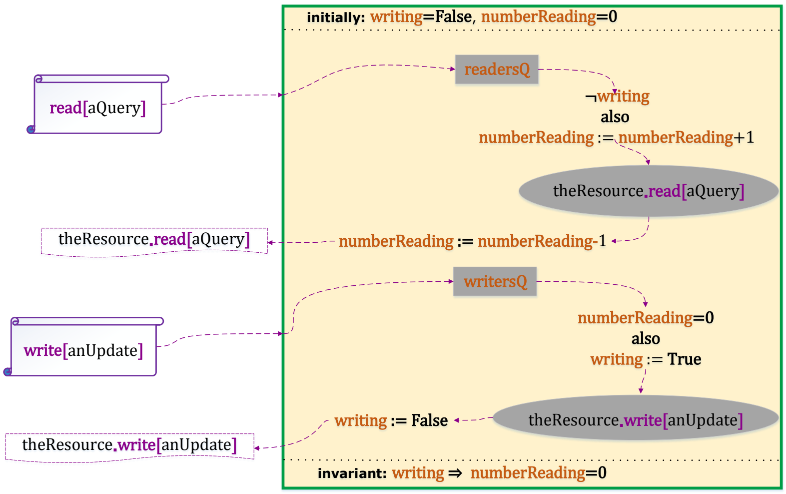
Swiss Cheese

Swiss cheese [Hewitt and Atkinson 1977, 1979; Atkinson 1980][16] is a programming language construct for scheduling concurrent access to shared resources with the following goals:

- *Generality:* Ability to conveniently program any scheduling policy
- *Performance:* Support maximum performance in implementation, *e.g*., the ability to avoid repeatedly recalculating conditions for proceeding.
- *Understandability:* Invariants of an Actor should hold at all observable execution points.

Concurrency control for readers and writers in a shared resource is a classic problem that illustrates limitations of Fog Cutter Actors. The fundamental constraint is that multiple writers are not allowed to operate concurrently and a writer is not allowed to operate concurrently with a reader.

Cheese diagram for **ReadersWriter** implementations:[i]



Note:

1. At most one activity is allowed to execute in the cheese.[ii]
2. The cheese has holes.[iii]
3. A variable can change only when in a continuous section of cheese.[iv]

Invariants hold at cheese boundaries, *i.e.*, an invariant must hold when the cheese is entered. Consequently, it doesn't matter what actions other Actors may be concurrently performing.

---

[i] The interface for the readers/writer guardian is the same as the interface for the shared resource:

Interface ReadersWriter with read[Query] → QueryAnswer¶
write[Update] → Void▮

[ii] Cheese is yellow in the diagram

[iii] A hole is grey in the diagram

[iv] Of course, other external Actors can change.

Futures

Futures [Baker and Hewitt 1977] are Actors that provide parallel execution by providing a proxy Actor for an expression while it is being computed.

Strong Types

Strong types play a crucial role in the Actor model for *equivalency* (e.g., Actor equivalence in pattern matching), *messaging* (sending and receiving messages), *encryption*, and *marshaling* (Actor communication between IoT devices).

Each type is an Actor, which in turn has a type. For example the type **Account** has type **Type**◁**Account**▷. A type governs equivalency of Actors of that type. For example if n and m are both of type **Float**, then equivalency of n and m is governed by sending **Float** an **equivalent?** message as follows: **Float**▪**equivalent?**[n, m]

Strong types must meet the following requirements:

- Each type is well-founded being constructed from primitive types, e.g., there is no type **Any**.[17]
- Every expression has a type and a variable always has the same type (across assignments).
- Types govern **Equivalence**, **Marshalling**, and **Messaging**.
- Casting is well-founded as follows:[18]
  - o Upcasting and downcasting for type extension and discrimination
  - o Internal casting by an Actor to one of its own interfaces (facets)

Speedy Contacting

Speedy Contracting aims to establish a legal contract between different parties over the Internet, with the entire process typically taking less than 1 second from start to finish.

The activity of a notary engaged (for contracting by a specified time) has two epochs:

A. **Before the notary decides that the time limit has passed:** for each signed copy of the contract (which specifies that it cannot be enforced without sending copies signed by all parties before the deadline) that the notary receives, it must respond with an acknowledgment (together with an ordered list of copies of all previous messages that the notary has sent and received) that says the following: The deposit was received within the time limit.

   If the notary has received signed copies from all parties, it must send a notice to each party (together with an ordered list of copies of all previous messages that the notary has sent and received) that provides the statement that all parties deposited signed copies of the contract within the time limit together with their signed copies.

B. **After the notary decides that the time limit has passed**: If it has not already sent notice that all parties have deposited signed copies of the contract before the deadline, it must send a notice to each party (together with an ordered list of copies of all previous messages that the notary has sent and received) that says the following: Not all parties deposited the signed contract within the time limit.

   Furthermore, for each signed copy of the contract that the notary receives, it must respond with an acknowledgment (together with an ordered list of copies of all previous messages that the notary has sent and received) that says the following: The deposit was *not* received within the time limit.

   Also, each party that receives notice from the notary that the all parties deposited signed copies before the deadline must forward it along with the signed copies to all other parties.

If a party has not received a status message from the notary or other parties significantly after the deadline, they can communicate with the notary and other parties asking for signed copies. If the party does not quickly receive copies signed by the notary or other parties, they can petition legal authorities to cancel the contract and that the government will hopefully respond in return for a fee.

Future work

As was the case with the λ-calculus,[i] it has taken decades since invention [Hewitt, Bishop, and Steiger 1973] to understand the scientific and engineering of Actor Systems and it is still very much a work in progress.

Actors are becoming the default model of computation. C#, Java, JavaScript, Objective C, and SystemVerilog are all headed in the direction of the Actor

[i] For example, it took over four decades to develop the eval message-passing model of the λ-calculus [Hewitt, Bishop, and Steiger 1973, Hewitt 2011] building on the Lisp procedural model.

Model and ActorScript is a natural extension of these languages. Since it is very close to practice, many programmers just naturally assume the Actor Model.

The following major developments in computer technology are pushing the Actor Model forward because Actor Systems are highly scalable:

- Many-core computer architectures
- Client-cloud computing

In fact, the Actor Model and ActorScript can be seen as codifying what are becoming some best programming practices for many-core and client-cloud computing.

Conclusion

The Actor Model is a mathematical theory that treats "*Actors*" as the universal conceptual primitives of concurrent digital computation. The model has been used both as a framework for a theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent systems. Unlike previous models of computation, the Actor Model was inspired by physical laws. It was also influenced by the programming languages Lisp, Simula-67 and Smalltalk-72, as well as ideas for Petri Nets, capabilities systems and packet switching. The advent of massive concurrency through client-cloud computing and many-core computer architectures has galvanized interest in the Actor Model.

When an Actor receives a message, it can concurrently:

- Send messages to (unforgeable) addresses of Actors that it has.
- Create new Actors.[i]
- Designate how to handle the next message received.

There is no assumed order to the above actions and they could be carried out concurrently. In addition two messages sent concurrently can be received in either order. Decoupling the sender from communication it sends was a fundamental advance of the Actor Model enabling asynchronous communication and control structures as patterns of passing messages.

Preferred methods for characterizing the Actor Model are as follows:

- *Axiomatically* stating laws that apply to all Actor *systems* [Baker and Hewitt 1977]
- *Denotationally* using the Computational Representation Theorem to characterize Actor computations [Clinger 1981; Hewitt 2006].
- O*perationally* using a suitable Actor programming language, *e.g.*, ActorScript [Hewitt 2012] that specifies how Actors can be implemented.

[i] with new addresses

The Actor Model can be used as a framework for modeling, understanding, and reasoning about, a wide range of concurrent systems.
For example:

- Electronic mail (e-mail) can be modeled as an Actor system. Accounts are modeled as Actors and email addresses as Actor addresses.
- Web Services can be modeled with endpoints modeled as Actor addresses.
- Objects with locks (e.g. as in Java and C#) can be modeled as Actors.
- The Actor Model can be a computational foundation for Inconsistency Robustness

Actor technology will see significant application for coordinating all kinds of digital information for individuals, groups, and institutions so their information usefully links together.

Information coordination needs to make use of the following information system principles:

- ***Persistence****. Information is collected and indexed.*
- ***Concurrency****: Work proceeds interactively and concurrently, overlapping in time.*
- ***Quasi-commutativity****: Information can be used regardless of whether it initiates new work or become relevant to ongoing work.*
- ***Sponsorship****: Sponsors provide resources for computation, i.e., processing, storage, and communications.*
- ***Pluralism****: Information is heterogeneous, overlapping and often inconsistent.*
- ***Provenance****: The provenance of information is carefully tracked and recorded*

The Actor Model is intended to provide a foundation for inconsistency robust information coordination.

Acknowledgements

Important contributions to the semantics of Actors have been made by: Gul Agha, Beppe Attardi, Henry Baker, Will Clinger, Irene Greif, Carl Manning, Ian Mason, Ugo Montanari, Maria Simi, Scott Smith, Carolyn Talcott, Prasanna Thati, and Aki Yonezawa.

Important contributions to the implementation of Actors have been made by: Gul Agha, Bill Athas, Russ Atkinson, Beppe Attardi, Henry Baker, Gerry Barber, Peter Bishop, Nanette Boden, Jean-Pierre Briot, Bill Dally, Blaine Garst, Peter de Jong, Jessie Dedecker, Ken Kahn, Rajesh Karmani, Henry Lieberman, Carl Manning, Mark S. Miller, Tom Reinhardt, Chuck Seitz, Amin Shali, Richard Steiger, Dan Theriault, Mario Tokoro, Darrell Woelk, and Carlos Varela.

Research on the Actor Model has been carried out at Caltech Computer Science, Kyoto University Tokoro Laboratory, MCC, MIT Artificial Intelligence Laboratory, SRI, Stanford University, University of Illinois at Urbana-Champaign Open Systems Laboratory, Pierre and Marie Curie University (University of Paris 6), University of Pisa, University of Tokyo Yonezawa Laboratory and elsewhere.

Conversations over the years with Dennis Allison, Bruce Anderson, Arvind, Bob Balzer, Bruce Baumgart, Gordon Bell, Dan Bobrow, Rod Burstall, Luca Cardelli, Vint Cerf, Keith Clark, Douglas Crockford, Jack Dennis, Peter Deutsch, Edsger Dijkstra, Scott Fahlman, Dan Friedman, Ole-Johan Dahl, Julian Davies, Patrick Dussud, Doug Englebart, Bob Filman, Kazuhiro Fuchi, Cordell Green, Jim Gray, Pat Hayes, Anders Hejlsberg, Pat Helland, John Hennessy, Tony Hoare, Mike Huhns, Dan Ingalls, Anita Jones, Bob Kahn, Gilles Kahn, Alan Karp, Alan Kay, Bob Kowalski, Monica Lam, Butler Lampson, Leslie Lamport, Peter Landin, Vic Lesser, Jerry Lettvin, Lick Licklider, Barbara Liskov, John McCarthy, Drew McDermott, Dave McQueen, Erik Meijer, Robin Milner, Marvin Minsky, Fanya S. Montalvo, Ike Nassi, Alan Newell, Kristen Nygaard, Seymour Papert, David Patterson, Carl Petri, Gordon Plotkin, Vaughan Pratt, John Reynolds, Jeff Rulifson, Earl Sacerdoti, Vijay Saraswat, Munindar Singh, Dana Scott, Ehud Shapiro, Burton Smith, Guy Steele, Gerry Sussman, Chuck Thacker, Kazunori Ueda, Dave Unger, Richard Waldinger, Peter Wegner, Richard Weyhrauch, Jeannette Wing, Terry Winograd, Glynn Winskel, David Wise, Bill Wulf, *etc*. greatly contributed to the development of the ideas in this article.

The members of the Silicon Valley Friday AM group made valuable suggestions for improving this paper. Blaine Garst found numerous bugs and made valuable comments including information on the historical development of interfaces. Patrick Beard found bugs and suggested improvements in presentation. Discussions with Dennis Allison, Eugene Miya, Vaughan Pratt and others were helpful in improving this article. As reviewers for Inconsistency Robustness 2011, Blaine Garst, Mike Huhns and Patrick Suppes made valuable suggestions for improvement. Discussions with Dale Schumacher helped clarify issues with Fog Cutter Actors and also helped debug the axiomatization of runtime failures in the Actor Model. Phil Bernstein, Sergey Bykov, and Gabi Kliot provide valuable comments on the section on Orleans Actors. Terry Hayes, Chris Hibbert, Daira Hopwood, Ken Kahn, Alan Karp, William Leslie, and Mark S. Miller and made helpful suggestions for the sections on capability, Orleans, and JavaScript Actors. Dan Ingalls made helpful suggestions on the sections on Smalltalk and elsewhere. Numerous typos were caught by Tom Pittard.

The Actor Model is intended to provide a foundation for scalable inconsistency-robust information coordination in privacy-friendly client-cloud computing [Hewitt 2009b].

Bibliography

Hal Abelson and Gerry *Sussman Structure and Interpretation of Computer Programs* 1984.

Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems* Doctoral Dissertation. 1986.

Gul Agha, Ian Mason, Scott Smith, and Carolyn Talcott. "A foundation for Actor computation." *Journal of Functional Programming*. 1997.

Mikael Amborn. *Facet-Oriented Program Design*. LiTH-IDA-EX–04/047–SE Linkőpings Universitet. 2004.

AMD *AMD64 Architecture Programmer's Manual* October 12, 3011

Joe Armstrong *History of Erlang* HOPL III. 2007.

Joe Armstrong. *Erlang*. CACM. September 2010.

William Athas and Charles Seitz *Multicomputers: message-passing concurrent computers* 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.

Russ Atkinson. *Automatic Verification of Serializers* MIT Doctoral Dissertation. June, 1980.

Henry Baker. *Actor Systems for Real-Time Computation* MIT EECS Doctoral Dissertation. January 1978.

Henry Baker and Carl Hewitt *The Incremental Garbage Collection of Processes* Proceeding of the Symposium on Artificial Intelligence Programming Languages. SIGPLAN Notices 12, August 1977.

Paul Baran. *On Distributed Communications Networks* IEEE Transactions on Communications Systems. March 1964.

Gerry Barber. *Reasoning about Change in Knowledgeable Office Systems* MIT EECS Doctoral Dissertation. August 1981.

John Barton. *Language Features* November 12, 2014. https://github.com/google/traceur-compiler/wiki/LanguageFeatures

Robert Bemer *How to consider a computer* Data Control Section, Automatic Control Magazine. March 1957.

Robert Bemer. *The status of automatic programming for scientific computation* Proc. 4th Annual Computer Applications Symposium. Armour Research Foundation. October 1057. (Panel discussion, pp. 118-126).

Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. *Orleans: Distributed Virtual Actors for Programmability and Scalabilit*y Microsoft MSR-TR-2014-41. March 24, 2014.

Harold Boley. *A Tight, Practical Integration of Relations and Functions* Springer. 1999.

Sergey Bykov, Alan Geller, Gabriel Kliot, James Larus, Ravi Pandya, and Jorgen Thelin. *Orleans: A Framework for Cloud Computing* Microsoft MSR-TR-2010-159. November 30, 2010

Sergey Bykov. *Building Real-Time Services for Halo* Microsoft Research. June 26, 2013.

Peter Bishop *Very Large Address Space Modularly Extensible Computer Systems* MIT EECS Doctoral Dissertation. June 1977.

Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman (2007a) *Interactive small-step algorithms I: Axiomatization* Logical Methods in Computer Science. 2007.

Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman (2007b*) Interactive small-step algorithms II: Abstract state machines and the characterization theorem.* Logical Methods in Computer Science. 2007.

W. Earl Boebert. *On the inability of an unmodified capability machine to enforce the *-property*. 7th DOD/NBS Computer Security Conference. September 24-26, 1984.

Per Brinch Hansen *Monitors and Concurrent Pascal: A Personal History* CACM 1996.

Stephen Brookes, Tony Hoare, and Bill Roscoe. *A theory of communicating sequential processes* JACM. July 1984.

Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Nielsen, Satish Thatte, Dave Winer. *Simple Object Access Protocol (SOAP) 1.1* World Wide Web Consortium Note. May 2000.

Jean-Pierre Briot. *Acttalk: A framework for object-oriented concurrent programming-design and experience* 2nd France-Japan workshop. 1999.

Jean-Pierre Briot. *From objects to Actors: Study of a limited symbiosis in Smalltalk-80* Rapport de Recherche 88-58, RXF-LITP. Paris, France. September 1988.

Mike Burnside, Dave Clarke, T. Mills, A. Maywah, S. Davadas, and Ronald Rivest. *Proxy-Based Security Protocols in Networked Mobile Devices*. SAC'2002.

Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson. *Modula-3 report (revised)* DEC Systems Research Center Research Report 52. November 1989.

Luca Cardelli and Andrew Gordon *Mobile Ambients* FoSSaCS'98.

Arnaud Carayol, Daniel Hirschkoff, and Davide Sangiorgi. *On the representation of McCarthy's amb in the π-calculus "*Theoretical Computer Science" February 2005.

Ajay Chander, Drew Dean, John Mitchell. *A State-Transition Model of Trust Management and Access Control. Proceedings of the Fourteenth IEEE Computer Security Foundations Workshop, Cape Breton, Nova Scotia, June 2001.*

Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. S*ynchronous, Asynchronous, and Causally Ordered Communication* Distributed Computing. 1995.

Alonzo Church "A Set of postulates for the foundation of logic (1&2)" Annals of Mathematics. Vol. 33, 1932. Vol. 34, 1933.

Alonzo Church. *An unsolvable problem of elementary number theory* American Journal of Mathematics. 58. 1936.

Alonzo Church *The Calculi of Lambda-Conversion* Princeton University Press. 1941.

Will Clinger. *Foundations of Actor Semantics* MIT Mathematics Doctoral Dissertation. June 1981.

Tyler Close. *Web-key: Mashing with Permission* WWW'08.

Melvin Conway. *Design of a separable transition-diagram compiler* CACM. 1963.

Eric Crahen. *Facet: A pattern for dynamic interfaces.* CSE Dept. SUNY at Buffalo. July 22, 2002.

Ole-Johan Dahl and Kristen Nygaard. "Class and subclass declarations" *IFIP TC2 Conference on Simulation Programming Languages.* May 1967.

Ole-Johan Dahl and Tony Hoare. *Hierarchical Program Structures* in "Structured Programming" Prentice Hall. 1972.

William Dally and Wills, D. *Universal mechanisms for concurrency* PARLE '89.

William Dally, et al. *The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms* IEEE Micro. April 1992.

Jack Dennis and Earl Van Horn. *Programming Semantics for Multiprogrammed Computations* CACM. March 1966.

ECMA. *C# Language Specification* June 2006.

ECMA *ECMAScript Language Specification 6th Edition Draft* December 6, 2014.

Jed Donnelley. *A Distributed Capability Computing System* Proceedings of the Third International Conference on Computer Communication. August, 1976.

Lars Ekeroth and Per-Martin Hedström. *General Packet Radio Service* (*GPRS) Support Notes* Ericsson Review No. 3. 2000.

Arthur Fine *The Shaky Game: Einstein Realism and the Quantum Theory* University of Chicago Press, Chicago, 1986.

Nissim Francez, Tony Hoare, Daniel Lehmann, and Willem-Paul de Roever. *Semantics of nondeterminism, concurrency, and communication* Journal of Computer and System Sciences. December 1979.

Christopher Fuchs *Quantum mechanics as quantum information (and only a little more)* in A. Khrenikov (ed.) Quantum Theory: Reconstruction of Foundations (Växjo: Växjo University Press, 2002).

Anna Funder. *Stasiland: Stories from Behind the Berlin Wall*. Harper. 2011.

Blaine Garst. *Origin of Interfaces* Email to Carl Hewitt on October 2, 2009.

Elihu M. Gerson. *Prematurity and Social Worlds* in Prematurity in Scientific Discovery. University of California Press. 2002.

Andreas Glausch and Wolfgang Reisig. *Distributed Abstract State Machines and Their Expressive Power* Informatik Berichete 196. Humboldt University of Berlin. January 2006.

Adele Goldberg and Alan Kay (ed.) *Smalltalk-72 Instruction Manual* SSL 76-6. Xerox PARC. March 1976.

Dina Goldin and Peter Wegner. *The Interactive Nature of Computing: Refuting the Strong Church-Turing Thesis* Minds and Machines March 2008.

Irene Greif and Carl Hewitt. *Actor Semantics of PLANNER-73* Conference Record of ACM Symposium on Principles of Programming Languages. January 1975.

Irene Greif. *Semantics of Communicating Parallel Professes* MIT EECS Doctoral Dissertation. August 1975.

Werner Heisenberg. *Physics and Beyond: Encounters and Conversations* translated by A. J. Pomerans (Harper & Row, New York, 1971), pp. 63– 64.

Carl Hewitt, Peter Bishop and Richard Steiger. *A Universal Modular Actor Formalism for Artificial Intelligence* IJCAI'73.

Carl Hewitt, *et al. Actor Induction and Meta-evaluation* Conference Record of ACM Symposium on Principles of Programming Languages, January 1974.

Carl Hewitt, *The Apiary Network Architecture for Knowledgeable Systems* Proceedings of Lisp Conference. 1980.

Carl Hewitt and Henry Lieberman. *Design Issues in Parallel Architecture for Artificial Intelligence* MIT AI memo 750. Nov. 1983.

Carl Hewitt, Tom Reinhardt, Gul Agha, and Giuseppe Attardi *Linguistic Support of Receptionists for Shared Resources* MIT AI Memo 781. Sept. 1984.

Carl Hewitt, *et al. Behavioral Semantics of Nonrecursive Control Structure* Proceedings of *Colloque sur la Programmation,* April 1974.

Carl Hewitt. *How to Use What You Know* IJCAI. September, 1975.

Carl Hewitt. *Viewing Control Structures as Patterns of Passing Messages* AI Memo 410. December 1976. Journal of Artificial Intelligence. June 1977.

Carl Hewitt and Henry Baker *Laws for Communicating Parallel Processes* IFIP-77, August 1977.

Carl Hewitt and Russ Atkinson. *Specification and Proof Techniques for Serializers* IEEE Journal on Software Engineering. January 1979.

Carl Hewitt, Beppe Attardi, and Henry Lieberman. *Delegation in Message Passing* Proceedings of First International Conference on Distributed Systems Huntsville, AL. October 1979.

Carl Hewitt and Gul Agha. *Guarded Horn clause languages: are they deductive and Logical?* in Artificial Intelligence at MIT, Vol. 2. MIT Press 1991.

Carl Hewitt and Jeff Inman. *DAI Betwixt and Between: From "Intelligent Agents" to Open Systems Science* IEEE Transactions on Systems, Man, and Cybernetics. Nov./Dec. 1991.

Carl Hewitt and Peter de Jong. *Analyzing the Roles of Descriptions and Actions in Open Systems* Proceedings of the National Conference on Artificial Intelligence. August 1983.

Carl Hewitt. (2006). "What is Commitment? Physical, Organizational, and Social" *COIN@AAMAS'06*. (Revised in Springer Verlag Lecture Notes in Artificial Intelligence. Edited by Javier Vázquez-Salceda and Pablo Noriega. 2007) April 2006.

Carl Hewitt (2007a). "Organizational Computing Requires Unstratified Paraconsistency and Reflection" *COIN@AAMAS*. 2007.

Carl Hewitt (2008a) *Norms and Commitment for iOrgs™* Information Systems: *Direct Logic™ and Participatory Argument Checking* ArXiv 0906.2756.

Carl Hewitt (2008b) "Large-scale Organizational Computing requires Unstratified Reflection and Strong Paraconsistency" *Coordination, Organizations, Institutions, and Norms in Agent Systems III* Jaime Sichman, Pablo Noriega, Julian Padget and Sascha Ossowski (ed.). Springer-Verlag. *http://organizational.carlhewitt.info/*

Carl Hewitt (2008c) *Middle History of Logic Programming: Resolution, Planner, Edinburgh Logic for Computable Functions, Prolog and the Japanese Fifth Generation Project* ArXiv 0904.3036.

Carl Hewitt (2008e). *ORGs for Scalable, Robust, Privacy-Friendly Client Cloud Computing* IEEE Internet Computing September/October 2008.

Carl Hewitt (2008f) *Formalizing common sense for scalable inconsistency-robust information integration using Direct Logic™* and the Actor Model Inconsistency Robust 2011.

Carl Hewitt (2009a) *Perfect Disruption: The Paradigm Shift from Mental Agents to ORGs* IEEE Internet Computing. Jan/Feb 2009.

Carl Hewitt (2009b) *A historical perspective on developing foundations for client-cloud computing: iConsult™ & iEntertain™ Apps using iInfo™ Information Integration for iOrgs™ Information Systems* (Revised version of "Development of Logic Programming: What went wrong, What was done about it, and What it might mean for the future" AAAI Workshop on What Went Wrong. AAAI-08.) ArXiv 0901.4934.

Carl Hewitt (2009c) *Middle History of Logic Programming: Resolution, Planner, Prolog and the Japanese Fifth Generation Project* ArXiv 0904.3036

Carl Hewitt (2010a) *ActorScript™ extension of C#®, Java®, and Objective C®:, iAdaptive™ concurrency for antiCloud™-privacy and security in* Inconsistency Robustness. College Publications. 2015.

Carl Hewitt, Erik Meijer, and Clemens 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.

Carl Hewitt. *"Health Information Systems Technologies" http://ee380.stanford.edu/cgi-bin/videologger.php?target=120606-ee380-300.asx* Slides for this video*:* *http://HIST.carlhewitt.info* Stanford CS Colloquium. June 6, 2012.

Carl Hewitt. *What is computation? Actor Model versus Turing's Model* in "A Computable Universe: Understanding Computation & Exploring Nature as Computation". edited by Hector Zenil. World Scientific Publishing. Company. 2012 . PDF at http://what-is-computation.carlhewitt.info

Carl Hewitt. *Security without IoT Mandatory Backdoors: Using Distributed Encrypted Public Recording to Catch & Prosecute Suspects.* HAL Archives. 2015/2016. https://hal.archives-ouvertes.fr/hal-01152495

Carl Hewitt. *All Writs is a Trojan Horse.* Letter to editor. CACM. May 2016.

Carl Hewitt. *Direct Logic for Intelligent Applications* Logic and Collaboration for Intelligent Applications. March 30-31, 2017.

Carl Hewitt. *Strong Types for Direct Logic* Logic and Collaboration for Intelligent Applications. March 30-31, 2017.

Carl Hewitt. *Axiomatics for Inconsistency Robust Direct Logic* Logic and Collaboration for Intelligent Applications. March 30-31, 2017.

Carl Hewitt, Dan Shapiro, Dan Flickinger, and Michael Mateas. *Building Infrastructure for Intelligent Applications* Logic and Collaboration for Intelligent Applications. March 30-31, 2017.

Tony Hoare *Quick sort.* Computer Journal 5 (1) 1962.

Tony Hoare *Monitors: An Operating System Structuring Concept* CACM. October 1974.

Tony Hoare. *Communicating sequential processes* CACM. August 1978.

Tony Hoare. *Communicating Sequential Processes* Prentice Hall. 1985.

Waldemer Horwat, Andrew Chien, and William Dally. *Experience with CST: Programming and Implementation* PLDI. 1989.

Daniel Ingalls. *Design Principles Behind Smalltalk*. Byte. August 1981.

Daniel Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. *Back to the Future: the story of Squeak, a practical Smalltalk written in itself* ACM Digital Library. 1997.

Peter Ingerman, *Thunks: A Way of Compiling Procedure Statements with Some Comments on Procedure Declarations*. CACM 4 (1). 1961.

Intel. *Intel Memory Protection Extensions (Intel MPX) support in the GCC compiler* GCC Wiki. November 24, 2014.

John Ioannidis. *Why Most Published Research Findings Are False* PLoS Medicine. 2(8): e124. 2005.

ISO. *ISO/IEC 14882:2011(E) Programming Languages -- C++, Third Edition* August, 2011.

Max Jammer *The EPR Problem in Its Historical Development* in Symposium on the Foundations of Modern Physics: 50 years of the Einstein-Podolsky-Rosen Gedankenexperiment, edited by P. Lahti and P. Mittelstaedt. World Scientific. Singapore. 1985.

Stanisław Jaśkowski On the Rules of Suppositions in Formal Logic *Studia Logica* 1, 1934. (reprinted in: *Polish logic 1920-1939*, Oxford University Press, 1967.)

Simon Peyton Jones, Andrew Gordon, Sigbjorn Finne. *Concurrent Haskell,* POPL'96.

Ken Kahn. *A Computational Theory of Animation* MIT EECS Doctoral Dissertation. August 1979.

Matthias Kaiser and Jens Lemcke *Towards a Framework for Policy-Oriented Enterprise Management* AAAI 2008.

Alan Karp, Rajiv Gupta, Guillermo Rozas, and Arindam Banerji. *Using Split Capabilities for Access Control* IEEE Software. January, 2003.

Alan Karp and Jun Li. *Solving the Transitive Access Problem for the Services Oriented Architecture* HPL-2008-204R1. HP Laboratories 2008.

Alan Karp and Jun Li. *Access Control for the Services Oriented Architecture* ACM Workshop on Secure Web Services. November 2007

Rajesh Karmani and Gul Agha. *Actors*. Encyclopedia of Parallel Computing 2011.

Alan Kay. *The early history of Smalltalk* ACM SIGPLAN Notices. Vol. 28. No. 3. March 1993.

Alan Kay. "Personal Computing" in *Meeting on 20 Years of Computing Science* Instituto di Elaborazione della Informazione, Pisa, Italy. 1975. http://www.mprove.de/diplom/gui/Kay75.pdf

Alan Kay. *Alan Kay on Messaging* Squeak email list. October 10, 1998.

Frederick Knabe *A Distributed Protocol for Channel-Based Communication with Choice* PARLE'92.

Jorgen Knudsen and Ole Madsen. *Teaching Object-Oriented Programming is more than teaching Object-Oriented Programming Languages* ECOOP'88. Springer. 1988.

Bill Kornfeld and Carl Hewitt. *The Scientific Community Metaphor* IEEE Transactions on Systems, Man, and Cybernetics. January 1981.

Bill Kornfeld. *Parallelism in Problem Solving* MIT EECS Doctoral Dissertation. August 1981.

Robert Kowalski. *A proof procedure using connection graphs* JACM. October 1975.

Robert Kowalski *Algorithm = Logic + Control* CACM. July 1979.

Robert Kowalski. *Response to questionnaire* Special Issue on Knowledge Representation. SIGART Newsletter. February 1980.

Robert Kowalski (1988a) *The Early Years of Logic Programming* CACM. January 1988.

Robert Kowalski (1988b) *Logic-based Open Systems* Representation and Reasoning. Stuttgart Conference Workshop on Discourse Representation, Dialogue tableaux and Logic Programming. 1988.

Stein Krogdahl. *The birth of Simula* HiNC 1 Conference. Trondheim. June 2003.

Albert Kwon, Udit Dhawan, Jonathan Smith, Tom. Knight, Jr., and André DeHon, "Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in 20th ACM Conference on Computer and Communications Security, November 2013.

Butler Lampson. *Protection*. In Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems. Princeton University, 1971.

Leslie Lamport *Time, Clocks, and Orderings of Events in a Distributed System* CACM. 1978.

Leslie Lamport *How to make a multiprocessor computer that correctly executes multiprocess programs* IEEE Transactions on Computers. 1979.

Peter Landin. *A Generalization of Jumps and Labels* UNIVAC Systems Programming Research Report. August 1965. (Reprinted in *Higher Order and Symbolic Computation.* 1998)

Peter Landin *A correspondence between ALGOL 60 and Church's lambda notation* CACM. August 1965.

Edward Lee and Stephen Neuendorffer. *Classes and Subclasses in Actor-Oriented Design*. Conference on Formal Methods and Models for Codesign (MEMOCODE). June 2004

Edward Lee. *Swarm Boxes*. SwarmLab UC Berkeley. March 19, 2015.

Henry Levy. Capability-Based Computer Systems Digital Press. 1985.

Steven Levy *Hackers: Heroes of the Computer Revolution* Doubleday. 1984.

Henry Lieberman. *An Object-Oriented Simulator for the Apiary* Conference of the American Association for Artificial Intelligence, Washington, D. C., August 1983

Henry Lieberman. *Thinking About Lots of Things at Once without Getting Confused: Parallelism in Act 1* MIT AI memo 626. May 1981.

Henry Lieberman. *A Preview of Act 1* MIT AI memo 625. June 1981.

Henry Lieberman and Carl Hewitt. *A real Time Garbage Collector Based on the Lifetimes of Objects* CACM June 1983.

Barbara Liskov *Data abstraction and hierarchy* Keynote address. OOPSLA'87.

Barbara Liskov and Liuba Shrira *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls* SIGPLAN'88.

Barbara Liskov and Jeannette Wing. *Behavioral subtyping using invariants and constraints* in "Formal methods for distributed processing: a survey of object-oriented approaches" Cambridge University Press. 2001.

Carl Manning. *Traveler: the Actor observatory* ECOOP 1987. Also appears in Lecture Notes in Computer Science, vol. 276.

Carl Manning,. *Acore: The Design of a Core Actor Language and its Compile* Master's Thesis. MIT EECS. May 1987.

Satoshi Matsuoka and Aki Yonezawa. *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages Research Directions in Concurrent Object-Oriented Programming* MIT Press. 1993.

John McCarthy *Programs with common sense* Symposium on Mechanization of Thought Processes. National Physical Laboratory, UK. Teddington, England. 1958.

John McCarthy. *A Basis for a Mathematical Theory of Computation* Western Joint Computer Conference. 1961.

John McCarthy, Paul Abrahams, Daniel Edwards, Timothy Hart, and Michael Levin. *Lisp 1.5 Programmer's Manual* MIT Computation Center and Research Laboratory of Electronics. 1962.

John McCarthy. *Situations, actions and causal laws* Technical Report Memo 2, Stanford University Artificial Intelligence Laboratory. 1963.

John McCarthy and Patrick Hayes. *Some Philosophical Problems from the Standpoint of Artificial Intelligence* Machine Intelligence 4. Edinburgh University Press. 1969.

Erik Meijer and Gavin Bierman. *A co-Relational Model of Data for Large Shared Data Banks* ACM Queue. March 2011.

Microsoft. *Asynchronous Programming with Async and Await* MSDN. 2013.
Giuseppe Milicia and Vladimiro Sassone. *The Inheritance Anomaly: Ten Years After* SAC. Nicosia, Cyprus. March 2004.
Mark S. Miller, Eric Dean Tribble, and Jonathan Shapiro. *Concurrency Among Strangers: Programming in E as Plan Coordination* Proceedings of the International Symposium on Trustworthy Global Computing. Springer. 2005.
Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. *Capability Myths Demolished* Submitted to Usenix Security 2003.
Mark S. Miller and Jonathan Shapiro *Paradigm Regained: Abstraction Mechanisms for Access Control* ASIAN'03. 2003.
Mark S. Miller *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control* Doctoral Dissertation. John Hopkins. 2006.
Mark S. Miller *et. al. Bringing Object-orientation to Security Programming*. YouTube. November 3, 2011.
Mark S. Miller. *mutable ArrayBuffer?* Communication on FriAM mailing list. May 14, 2016.
George Milne and Robin Milner. "Concurrent processes and their syntax" *JACM*. April, 1979.
Robert Milne and Christopher Strachey. *A Theory of Programming Language Semantics* Chapman and Hall. 1976.
Robin Milner *Processes: A Mathematical Model of Computing Agents* Proceedings of Bristol Logic Colloquium. 1973.
Robin Milner *Elements of interaction: Turing award lecture* CACM. January 1993.
Marvin Minsky (ed.) *Semantic Information Processing* MIT Press. 1968.
Ugo Montanari and Carolyn Talcott. *Can Actors and Pi-Agents Live Together?* Electronic Notes in Theoretical Computer Science. 1998.
Eugenio Moggi *Computational lambda-calculus and monads* IEEE Symposium on Logic in Computer Science. Asilomar, California, June 1989.
Allen Newell and Herbert Simon. *The Logic Theory Machine: A Complex Information Processing System.* Rand Technical Report P-868. June 15, 1956
Kristen Nygaard. *SIMULA: An Extension of ALGOL to the Description of Discrete-Event Networks* IFIP'62. Kristen Nygaard. *Basic Concepts in Object Oriented Programming* Special Edition on Object-Oriented Programming Languages. SIGPLAN Notices. Vol. 21. November 1986.
David Park. *Concurrency and Automata on Infinite Sequences* Lecture Notes in Computer Science, Vol. 104. Springer. 1980.
Elliot Organick. *A Programmer's View of the Intel 432 System* McGraw-Hill, 1983.
Carl Petri. *Kommunikation mit Automate* Ph. D. Thesis. University of Bonn. 1962.
Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. *A semantics for imprecise exceptions* Conference on Programming Language Design and Implementation. 1999.

Gordon Plotkin. *A powerdomain construction* SIAM Journal of Computing. September 1976.

George Polya (1957) *Mathematical Discovery: On Understanding, Learning and Teaching Problem Solving Combined Edition* Wiley. 1981.

Karl Popper (1935, 1963) *Conjectures and Refutations: The Growth of Scientific Knowledge* Routledge. 2002.

Claudius Ptolemaeus, Editor. *System Design, Modeling, and Simulation: Using Ptolemy II* LuLu. http://ptolemy.org/systems. 2014

Susan Rajunas. *The KeyKOS/KeySAFE System Design.* Technical Report SEC009-01. Key Logic, Inc. March 1989.

John Reppy, Claudio Russo, and Yingqi Xiao *Parallel Concurrent ML* ICFP'09.

John Reynolds. *Definitional interpreters for higher order programming languages* ACM Conference Proceedings. 1972.

John Reynolds. *The Discoveries of Continuations* Lisp and Symbolic Computation *6 (3-4).* 1993.

Bill Roscoe. *The Theory and Practice of Concurrency* Prentice-Hall. Revised 2005.

Alex Russell. *A design for Futures/Promises in DOM.* W3C. March 7, 2013.

Dale Schumacher. *Implementing Actors in Kernel* February 16, 2012. http://www.dalnefre.com/wp/2012/02/implementing-actors-in-kernel/

Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages* Oxford Programming Research Group Technical Monograph. PRG-6. 1971

Dana Scott *Data Types as Lattices.* SIAM Journal on computing. 1976.

Charles Seitz. *The Cosmic Cube* CACM. Jan. 1985.

Peter Sewell, et. al. *x86-TSO: A Rigorous and Usable Programmer's Model for x86 Microprocessors* CACM. July 2010.

Jonathan Shapiro and Jonathan Adams. *Coyotos Microkernel Specification* EROS Group. September 10, 2007.

SIGPLAN. *Special Edition on Object-Oriented Programming Languages* SIGPLAN Notices. Vol. 21. November 1986.

Michael Smyth. *Power domains.* Computer and System Sciences. 1978.

Alfred Spiessens. *Patterns of Safe Collaboration.* Doctoral Thesis. Université catholique de Louvain. February 2007.

Guy Steele Jr. *Lambda: The Ultimate Declarative* MIT AI Memo 379. November 1976.

Lynn Stein, Henry Lieberman, and David Ungar. *A Shared View of Sharing: The Treaty of Orlando* Object-oriented concepts, databases, and applications. ACM. 1989

Jan Stenberg. *Building Halo 4, a Video Game, Using Actor Model.* InfoQ. March 7, 2015.

Gunther Stent. *Prematurity and Uniqueness in Scientific Discovery* Scientific American. December, 1972.

Sun *Java Specification Request 133* 2004

Gerry Sussman and Guy Steele *Scheme: An Interpreter for Extended Lambda Calculus* AI Memo 349. December, 1975.

David Suzuki. *The Thrill of Seeing Ants for What They* are The David Suzuki Reader. Greystone Books. 2014.

Daniel Theriault. *A Primer for the Act-1 Language* MIT AI memo 672. 1982.

Daniel Theriault. *Issues in the Design and Implementation of Act 2* MIT AI technical report 728. June 1983.

Hayo Thielecke *An Introduction to Landin's "A Generalization of Jumps and Labels"* Higher-Order and Symbolic Computation. 1998.

Dave Thomas and Brian Barry. *Using Active Objects for Structuring Service Oriented Architectures: Anthropomorphic Programming with Actors* Journal of Object Technology. July-August 2004.

Bill Tulloh and Mark S. Miller. *Institutions as Abstraction Boundaries* Humane Economics: Essays in Honor of Don Lavoie. Elgar Publishing. 2006.

Vaughan Vernon. *Reactive Messaging Patterns with the Actor Model.* Pearson Education. 2016.

Ulf Wiger. *1000 Year-old Design Patterns* QCon. Apr 21, 2011.

Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. *The CHERI capability model: Revisiting RISC in an age of risk* International Symposium on Computer Architecture (ISCA). June 2014.

Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. *CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization* IEEE Symposium on Security and Privacy. May 2015

Darrell Woelk. *Developing InfoSleuth Agents Using Rosette: An Actor Based Language* Proceedings of the CIKM '95 Workshop on Intelligent Information Agents. 1995.

World Wide Web Consortium. *HTML5: A vocabulary and associated APIs for HTML and XHTML* Editor's Draft. August 22, 2012.

Akinori Yonezawa, Ed. *ABCL: An Object-Oriented Concurrent System* MIT Press. 1990.

Akinori Yonezawa *Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics* MIT EECS Doctoral Dissertation. December 1977.

Takeshi Yoshino. *Intent to Implement: DOM Futures.* W3C. May 23, 2013

Hadasa Zuckerman and Joshua Lederberg. *Postmature Scientific Discovery?* Nature. December, 1986.

Appendix 1. Historical background[19]

The Actor Model builds on previous models of nondeterministic computation. Several models of nondeterministic computation were developed including the following:

Concurrency versus Turing's Model

Turing's model of computation was intensely psychological.[20] [Sieg 2008] formalized it as follows:

- *Boundedness:* A computer can immediately recognize only a bounded number of configurations.
- *Locality:* A computer can change only immediately recognizable configurations.

In the above, computation is conceived as being carried out in a single place by a device that proceeds from one well-defined state to the next.

Computations are represented differently in Turing Machines and Actors:

1. *Turing Machine*: a computation can be represented as a global state that determines all information about the computation.[21] It can be nondeterministic as to which will be the next global state.
2. *Actors*: a computation can be represented as a configuration. Information about a configuration can be indeterminate.[i]

λ-calculus

The λ-calculus was originally developed as part of a system for the foundations of logic [Church 1932-33]. However, the system was soon shown to be inconsistent. Subsequently, Church removed logical propositions from the system leaving a purely procedural λ-calculus [Church 1941].[22]

However, the semantics of the λ-calculus were expressed using string substitution in which the values of parameters were substituted into the body of an invoked λ-expression. The substitution model is unsuitable for concurrency because it does not allow the capability of sharing of changing resources.

That Actors, which behave like mathematical functions, exactly correspond with those definable in the λ-calculus provides an intuitive justification for the rules of the λ-calculus:

- *Identifiers*: each identifier is bound to the address of an Actor. The rules for free and bound identifiers correspond to the Actor rules for addresses.
- *Beta reduction*: each beta reduction corresponds to an Actor receiving a message. Instead of performing substitution, an Actor receives addresses of its arguments.

---

[i] For example, there can be messages in transit that will be delivered at some indefinite time.

Inspired by the λ-calculus, the interpreter for the programming language Lisp [McCarthy *et. al.* 1962] made use of a data structure called an environment so that the values of parameters did not have to be substituted into the body of an invoked λ-expression.[23]

Note that in the definition in ActorScript [Hewitt 2011] of the λ-calculus below:

- All operations are local.
- The definition is modular in that each λ-calculus programming language construct is an Actor.
- The definition is easily extensible since it is easy to add additional programming language constructs.
- The definition is easily operationalized into efficient concurrent implementations.
- The definition easily fits into more general concurrent computational frameworks for many-core and distributed computation

The λ-calculus can be implemented in ActorScript as follows:

```
Actor Identifier◁aType▷[aString:String]
  implements Expression◁aType▷ using
    eval[e:Environment]:aType ∴  e▪lookup[⊡Identifier◁aType▷]
                                   // lookup this identifier in environment e

Actor ProcedureCall◁aType, anotherType▷
                        [operator:Expression◁[aType]↦ anotherType▷,
                         operand:Expression◁aType▷]
  implements Expression◁anotherType▷ using
    eval[e:Environment]:anotherType ∴
         (operator.eval[e])▪[operand▪eval[e]]

Actor Lambda◁aType, anotherType▷
         [anIdentifier:Identifier◁aType▷,
          body:Expression◁anotherType▷]
  implements Expression◁[aType]→anotherType▷ using
    eval[e:Environment]:anotherType ∴
      λ [anArgument:aType]
         body▪eval[e▪bind[anIdentifier, anArgument]]
             // eval body in a new environment
                 //with anIdentifier bound to anArgument
                       // as extension of environment e
```

In many practical applications, the parallel λ-calculus and pure Logic Programs can be thousands of times slower than Actor implementations.[i]

Petri nets

Prior to the development of the Actor Model, Petri nets[24] were widely used to model nondeterministic computation. However, they were widely acknowledged to have an important limitation: they modeled control flow but not data flow. Consequently they were not readily composable thereby limiting their modularity.

Hewitt pointed out another difficulty with Petri nets:

> Simultaneous action, *i.e.,* the atomic step of computation in Petri nets is a transition in which tokens simultaneously disappear from the input places of a transition and appear in the output places. The physical basis of using a primitive computational entity with this kind of simultaneity seemed questionable to him.

Despite these apparent difficulties, Petri nets continue to be a popular approach to modeling nondeterminism, and are still the subject of active research.

Simula

Simula 1 [Nygaard 1962] pioneered nondeterministic discrete event simulation using a global clock:

> In this early version of Simula a system was modeled by a (fixed) number of "stations", each with a queue of "customers". The stations were the active parts, and each was controlled by a program that could "input" a customer from the station's queue, update variables (global, local in station, and local in customer), and transfer the customer to the queue of another station. Stations could discard customers by not transferring them to another queue, and could generate new customers. They could also wait a given period (in simulated time) before starting the next action. Custom types were declared as data records, without any actions (or procedures) of their own.
> [Krogdahl 2003]

Thus at each time step, the program of the next station to be simulated would update the variables.

Kristen Nygaard and Ole-Johan Dahl developed the idea (first described in an IFIP workshop in 1967) of organizing objects into "classes" with "subclasses" that could inherit methods for performing operations from their super classes. In this way, Simula-67 considerably improved the modularity of nondeterministic discrete event simulations.

[i] For example, implementations using Actors of Direct Logic can be thousands of times faster than implementations in the parallel λ-calculus.

According to [Krogdahl 2003]:

> *Objects could act as processes that can execute in "quasi-parallel" that is in fact a form of nondeterministic sequential execution in which a simulation is organized as "independent" processes. Classes in Simula 67 have their own procedures that start when an object is generated. However, unlike Algol procedures, objects may choose to temporarily stop their execution and transfer the control to another process. If the control is later given back to the object, it will resume execution where the control last left off. A process will always retain the execution control until it explicitly gives it away. When the execution of an object reaches the end of its statements, it will become "terminated", and can no longer be resumed (but local data and local procedures can still be accessed from outside the object).*
>
> *The quasi-parallel sequencing is essential for the simulation mechanism. Roughly speaking, it works as follows: When a process has finished the actions to be performed at a certain point in simulated time, it decides when (again in simulated time) it wants the control back, and stores this in a local "next-event-time" variable. It then gives the control to a central "time-manager", which finds the process that is to execute next (the one with the smallest next-event-time), updates the global time variable accordingly, and gives the control to that process.*
>
> *The idea of this mechanism was to invite the programmer of a simulation program to model the underlying system by a set of processes, each describing some natural sequence of events in that system (e.g. the sequence of events experienced by one car in a traffic simulation).*
>
> *Note that a process may transfer control to another process even if it is currently inside one or more procedure calls. Thus, each quasi-parallel process will have its own stack of procedure calls, and if it is not executing, its "reactivation point" will reside in the innermost of these calls. Quasi-parallel sequencing is analogous to the notion of co-routines [Conway 1963].*

Note that Simula-67 operated on the global state of a simulation of which the local variables of simulated objects formed state components.[25] Simulated objects were conceived as abstract data types in a class hierarchy that were operated on by virtual procedures [Dahl and Hoare 1972] rather than as independent Actors asynchronously sent messages. Also Simula-67 lacked formal interfaces and instead relied on inheritance in a hierarchy of objects thereby placing limitations to the ability to define and invoke behavior not directly inherited.

Types in Simula-67 are the names of implementations called "classes" in contrast with ActorScript in which types are interfaces that do not name their implementation. Also, although Simula-67 had nondeterminism, it did not have concurrency.[26]

Planner

The two major paradigms for constructing semantic software systems were procedural and logical. The procedural paradigm was epitomized by using Lisp [McCarthy *et al.* 1962; Minsky, *et al.* 1968] recursive procedures operating on list structures. The logical paradigm was epitomized by uniform resolution theorem provers [Robinson 1965].

Planner [Hewitt 1969] was a kind of hybrid between the procedural and logical paradigms.[27] An implication of the form (P *implies* Q) was procedurally interpreted as follows:[28]

- **When asserted** P**, Assert** Q
- **When goal** Q**, SetGoal** P
- **When asserted** (*not* Q)**, Assert** (*not* P)
- **When goal** (*not* P)**, SetGoal** (*not* Q)

Planner was the first programming language based on the pattern-directed invocation of procedural plans from assertions and goals. ***It represented a rejection of the resolution uniform proof procedure paradigm.***

Smalltalk-72

Alan Kay was influenced by message passing in the pattern-directed invocation of Planner in developing Smalltalk-71. Hewitt was intrigued by Smalltalk-71 but was put off by the complexity of communication that included invocations with many fields including the following:[29]

Global	the environment of the parameter values
SENDER	the sender of the message
REPLY-STYLE	the receiver of the message
STATUS	wait, fork, ...?
REPLY	progress of the message
OPERATION SELECTOR	relative to the receiver
# OF PARAMETERS	
PARAMETER_1	
...	
PARAMETER_N	

In November 1972, Kay visited MIT and presented a lecture on some of his ideas for Smalltalk-72 building on the Logo work of Seymour Papert and the "little person" metaphor of computation used for teaching children to program. Smalltalk-72 made important advances in graphical user interfaces.

However, the message passing of Smalltalk-72 was quite complex [Kay 1975]. Code in the language was viewed by the interpreter as simply a stream of tokens. According to [Ingalls 1983]:[30]

> *The first (token) encountered (in a program) was looked up in the dynamic context, to determine the receiver of the subsequent message. The name lookup began with the class dictionary of the current activation. Failing there, it moved to the sender of that activation and so on up the sender chain. When a binding was finally found for the token, its value became the receiver of a new message, and the interpreter activated the code for that object's class.*[31]

Thus the message passing model in Smalltalk-72 was closely tied to a particular machine model and programming language syntax that did not lend themselves to concurrency. `SENDER` was retained as part of the message-passing protocol, which is problematical for distributed systems. Also, although the system was bootstrapped on itself, the behavior of language constructs was defined (like Lisp) by an interpreter instead by their response to `eval` messages.

Planner, Simula-67, Smalltalk-72 [Kay 1975; Ingalls 1983] and packet-switched networks had previously used message passing. However, they were too complicated to use as the foundation for a mathematical theory of computation. Also they did not address fundamental issues of concurrency.

Actors

The invention of digital computers caused a decisive paradigm shift when the notion of an interrupt was invented so that input that is received asynchronously from outside could be incorporated in an ongoing computation. At first concurrency was conceived using low level machine implementation concepts like threads, locks, coherent memory, channels, cores, queues, *etc.*

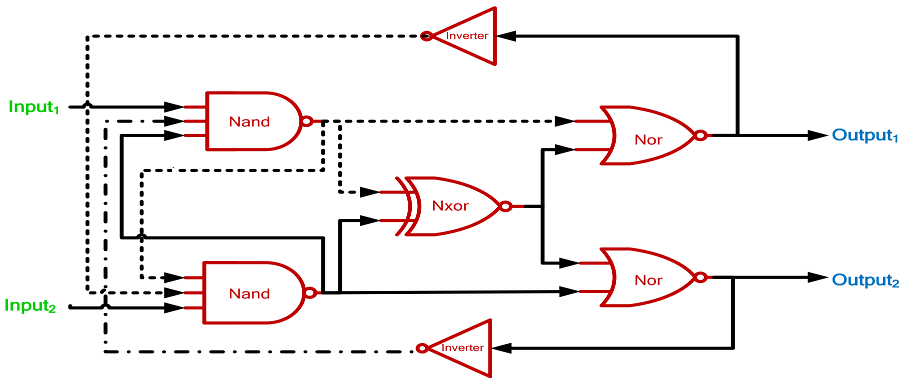
The Actor Model [Hewitt, Bishop, and Steiger 1973; *etc*.] was based on message passing that was different from previous models of computation because the sender of a message is not intrinsic to the semantics of a communication.[32]

In contrast to previous global state model, computation in the Actor Model is conceived as distributed in space where computational devices called Actors communicate asynchronously using addresses of Actors and the entire computation is not in any well-defined state.[33]

Axioms of locality including *Structural* and *Operational* hold as follows:

- *Structural:* The local storage of an Actor can include *addresses* only
    1. that were provided when it was created
    2. that have been received in messages
    3. that are for Actors created here
- *Operational:* In response to a message received, an Actor can
    1. create more Actors
    2. send messages[i] to *addresses* in the following:
        - the message it has just received
        - its local storage
    3. designate how to process the next message received

In concrete terms for Actor systems, typically we cannot observe the details by which the order in which an Actor processes messages has been determined. Attempting to do so affects the results. Instead of observing the internals of arbitration processes of Actor computations, we await outcomes.[34] Indeterminacy in arbiters produces indeterminacy in Actors.[ii]



Arbiter Concurrency Primitive[35]

After the above circuit is started, it can remain in a meta-stable state for an unbounded period of time before it finally asserts either $Output_1$ or $Output_2$. So there is an inconsistency between the nondeterministic state model of computation and the circuit model of arbiters.[36]

The internal processes of arbiters are not public processes. Attempting to observe them affects their outcomes. Instead of observing the internals of arbitration processes, we necessarily await outcomes. Indeterminacy in arbiters produces indeterminacy in Actors. The reason that we await outcomes is that we have no realistic alternative.

---

[i] Likewise the messages sent can contain addresses only
1. that were provided when the Actor was created
2. that have been received in messages
3. that are for Actors created here

[ii] The dashed lines are used only to disambiguate crossing wires.

The Actor Model integrated the following:

- the λ-calculus
- interrupts
- blocking method invocation
- imperative programming using locks
- capabilities systems
- co-routines
- packet networks
- email systems
- Petri nets
- Smalltalk-72
- Simula-67
- pattern-directed invocation (from Planner)

In 1975, Irene Greif published the first operational model of Actors in her dissertation. Two years after Greif published her operational model, Carl Hewitt and Henry Baker published the Laws for Actors [Baker and Hewitt 1977].

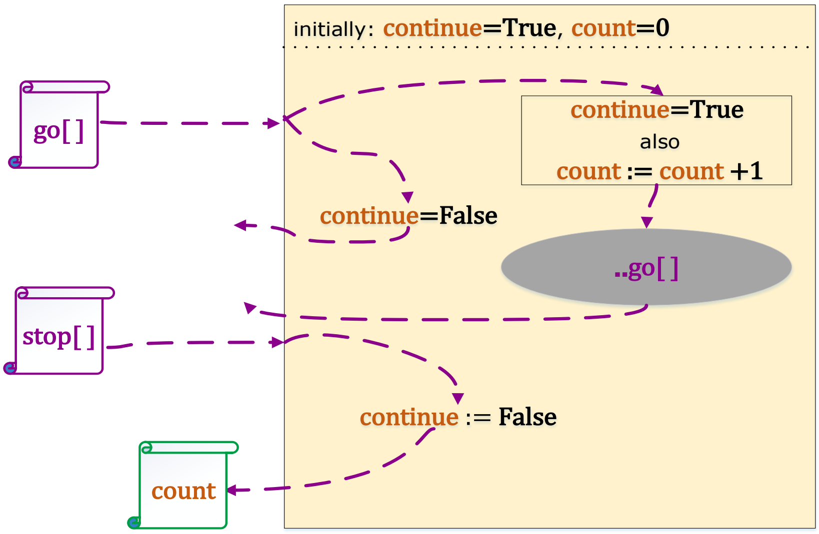
Indeterminacy in Concurrent Computation

The first models of computation (*e.g.* Turing machines, Post productions, the λ-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]). 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.[37] 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.[38]

Gordon Plotkin [1976] gave an informal proof as follows:

> *Now the set of initial segments of execution sequences of a given nondeterministic program* P*, starting from a given state, will form a tree. The branching points will correspond to the choice points in the program. Since there are always only finitely many alternatives at each choice point, the branching factor of the tree is always finite.*[39] *That is, the tree is finitary. Now König's lemma says that if every branch of a finitary tree is finite, then so is the tree itself. In the present case this means that if every execution sequence of* P *terminates, then there are only finitely many execution sequences. So if an output set of* P *is infinite, it must contain a nonterminating computation*.[40]

The above proof is quite general and applies to the Abstract State Machine (ASM) model [Blass, Gurevich, Rosenzweig, and Rossman 2007a, 2007b; Glausch and Reisig 2006], which consequently are not really models of concurrency. It also applies to the parallel λ-calculus, which includes all the capabilities of the nondeterministic λ-calculus. Researchers (before the Actor Model was invented) hypothesized that the parallel λ-calculus naturally modeled all of computation and their research programme was to reduce all computation to the parallel λ-calculus [Scott and Strachey 1971, Milne and Strachey 1976]. One of the important early discoveries in the development of the Actor Model was that all of computation is not reducible to the parallel λ-calculus. In fact, there are Actor computations that cannot be implemented in the parallel λ-calculus. For example, by the semantics of the Actor Model of computation [Clinger 1981] [Hewitt 2006], concurrently sending the Actor below both a **start** message and a **stop** message will result in returning an integer of unbounded size for the **stop** message.



Theorem. There are nondeterministic computable functions on integers that cannot be implemented by a nondeterministic Turing machine.

Proof. The above Actor system implements a nondeterministic function[i] that cannot be implemented by a nondeterministic Turing machine.

Consequently, the above concurrent algorithm for Unbounded.**[ ]** cannot be implemented using nondeterministic abstract state machines or using the nondeterministic λ-calculus.

[i] with graph {**[ ]**↝**0**, **[ ]**↝**1**, **[ ]**↝**2**, **…**}

Nondeterminism is a special case of Indeterminism.

Consider the following Nondeterministic Turing Machine that starts at *Step 1*:

Step 1: Either print 1 on the next square of tape or execute *Step 3.*
Step 2: Execute *Step 1.*
Step 3: Halt.

According to the definition of Nondeterministic Turing Machines, the above machine might never halt.

Note that the computations performed by the above machine are structurally different than the computations performed by the above counter Actor in the following way:

1. The decision making of the above Nondeterministic Turing Machine is internal (having an essentially psychological basis).
2. The decision making of the above counter Actor exhibits physical indeterminacy.

Edsger Dijkstra further developed the nondeterministic global state approach, which gave rise to a controversy concerning *unbounded nondeterminism*[i]. Unbounded nondeterminism is 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 providing a guarantee that the request will be serviced*. The Actor Model provides 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]. He believed that it was impossible to implement unbounded nondeterminism.

Computation is not subsumed by logical deduction

Kowalski claims that "*computation could be subsumed by deduction"*[41] The gauntlet was officially thrown in *The Challenge of Open Systems* [Hewitt 1985] to which [Kowalski 1988b] replied in *Logic-Based Open Systems.* [ii] This was followed up with [Hewitt and Agha 1988] in the context of the Japanese Fifth Generation Project.

According to Hewitt, *et. al.* and contrary to Kowalski computation in general cannot be subsumed by deduction and contrary to the quotation (above)

---

[i] A system is defined to have *unbounded nondeterminism* exactly when both of the following hold:

1. When started, the system always halts.
2. For every integer n, the system can halt with an output that is greater than n.

[ii] [Kowalski 1979] forcefully stated:

There is only one language suitable for representing information -- whether declarative or procedural -- and that is first-order predicate logic. There is only one intelligent way to process information -- and that is by applying deductive inference methods.

attributed to Hayes computation in general is not subsumed by deduction. [Hewitt and Agha 1991] and other published work argued that mathematical models of concurrency did not determine particular concurrent computations because they make use of arbitration for determining the order in which messages are processed. These orderings cannot be deduced from prior information by mathematical logic alone. Therefore mathematical logic cannot implement concurrent computation in open systems.

A nondeterministic system is defined to have "*unbounded nondeterminism*"[i] exactly when both of the following hold:

1. When started, the system *always* halts.
2. For every integer n, it is possible for the system to halt with output that is greater than n.

This article has discussed the following points about unbounded nondeterminism controversy:

- A Nondeterministic Turing Machine cannot implement unbounded nondeterminism.
- A pure Logic Program[42] cannot implement unbounded nondeterminism.
- Semantics of unbounded nondeterminism are required to prove that a server provides service to every client.[43]
- An Actor system [Hewitt, *et. al*. 1973] can implement servers that provide service to every client and consequently unbounded nondeterminism.
- Dijkstra believed that unbounded nondeterminism cannot be implemented [Dijkstra 1967; Dijkstra and van Gasteren 1986].
- The semantics of CSP [Francez, Hoare, Lehmann, and de Roever 1979] specified bounded nondeterminism for reasons mentioned above in the article. Since Hoare *et. al.* wanted to be able to prove that a server provided service to clients, the semantics of a subsequent version of CSP were switched from bounded to unbounded nondeterminism.
- Unbounded nondeterminism was but a symptom of deeper underlying issues with sequential processes using nondeterministic global states as a foundation for computation.[ii]

The Computational Representation Theorem [Clinger 1981, Hewitt 2006] characterizes the semantics of Actor Systems without making use of sequential processes.

[i] For example the following systems do *not* have unbounded nondeterminism:
- A nondeterministic system which sometimes halts and sometimes doesn't
- A nondeterministic system that always halts with an output less than 100,000.
- An operating system that never halts.

[ii] See [Knabe 1992].

Actor Model versus Classical Objects

The following are fundamental differences between the Actor Model and Classical Objects[Nygaard and Dahl 1967, Nygaard 1986]:

- Classical Objects[44] are founded on "a physical model, simulating the behavior of either a real or imaginary part of the world"[45], whereas the Actor Model is founded on the physics of computation.
- Every Classical Object[46] is an instance of a Class[i] in a hierarchy[47], whereas an Actor can implement multiple interfaces.[48]
- Virtual Procedures can be used to operate on Objects, whereas messages[ii] can be sent to Actors.[49]

Unfortunately, Objects remain ill-defined. Consequently, the term "Object" bas been used in inconsistent ways in the literature.

Hairy Control Structure

Peter Landin introduced a powerful co-routine control structure using his **J** (for Jump) operator that could perform a nonlocal goto into the middle of a procedure invocation [Landin 1965]. In fact the **J** operator enabled a program to jump back into the middle of a procedure invocation even after it had already returned!

[Reynolds 1972] introduced control structure continuations using a construct called **escape** that is a more structured versions of Landin's **J** operator. Sussman and Steele called their variant of **escape** by the name "*call with current continuation.*" Using *escape* can leave Actor customers stranded. Consequently, use of **escape** is generally avoided these days and exceptions[50] are used instead so that clean up can be performed.

Using the **J** operator, McDermott, and Sussman [1972] developed the Lisp-based language Conniver based on "hairy control structure" that could implement non-chronological backtracking that was more general than the chronological backtracking in Planner. However, hairy control structure did not work out well in practice because it was very difficult to understand and debug procedures that could return more than once.

Pat Hayes remarked:

> *Their* [Sussman and McDermott] *solution, to give the user access to the implementation primitives of Planner, is however, something of a retrograde step (what are Conniver's semantics?).* [Hayes 1974]

---

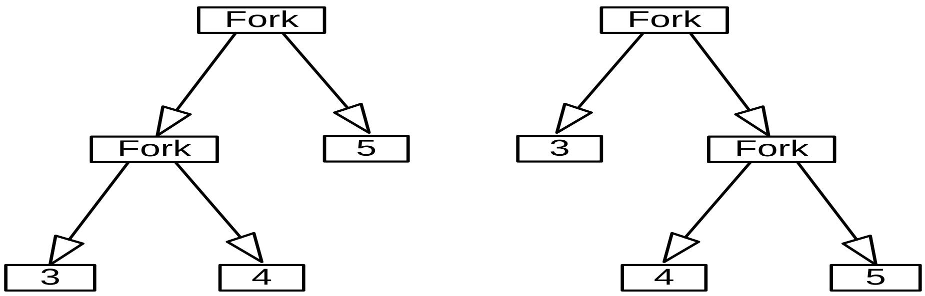
[i] A Class is an implementation of an Actor.

[ii] A message can be one-way and each must be of type **Message**.

Hewitt had concluded:

> *One of the most important results that has emerged from the development of Actor semantics has been the further development of techniques to semantically analyze or synthesize control structures as patterns of passing messages.* ***As a result of this work, we have found that we can do without the paraphernalia of "hairy control structure."*** [51](emphasis in original)

For example, futures can be adaptively created to do the kind of computation performed by hairy structure. [Hewitt 1974] invented the same-fringe problem as an illustration where the "fringe" of a tree is a list of all the leaf nodes of the tree.



Two trees with the same fringe [3 4 5]

Below is the definition of a procedure that computes a List that is the "fringe" of the leaves of tree.

```
Fringe.[aTree:Tree]:List ≡
   aTree � Leaf[x] ⦂ [x]
           Fork[tree1, tree2] ⦂
              [∨Fringe.[tree1], ∨Fringe.[tree2]]
```

The above procedure can be used to define SameFringe that determines if two lists have the same fringe [Hewitt 1972]:

```
SameFringe.[aTree:Tree, anotherTree:Tree]:Boolean ≡
                                        //  test if two trees have the same fringe
   Fringe.[aTree]=Fringe.[anotherTree]▮
```

Using Actors in this way obviates the need for explicit co-routine constructs, *e.g.*, *yield* in C# [ECMA 2006], JavaScript [ECMA 2014], *etc*.

In the 1960's at the MIT AI Lab a remarkable culture grew up around "*hacking*" that concentrated on remarkable feats of programming.[52] Growing out of this tradition, Gerry Sussman and Guy Steele decided to try to understand Actors by reducing them to machine code that they could understand and so developed a "*Lisp-like language, Scheme, based on the lambda calculus, but extended for side effects, multiprocessing, and process synchronization*." [Sussman and Steele 1975].

Their reductionist approach included primitives like the following: START!PROCESS, STOP!PROCESS, and EVALUATE!UNINTERRUPTIBLEY.[i] Of course, the above reductionist approach is unsatisfactory because it missed a crucial aspect of the Actor Model: the reception ordering of messages.

Sussman and Steele [1975] noticed some similarities between Actor programs and the λ-calculus. They mistakenly concluded that they had reduced Actor programs to a "continuation-passing programming style":

> *It is always possible, if we are willing to specify explicitly what to do with the answer, to perform any calculation in this way: rather than reducing to its value, it reduces to an application of a continuation to its value. That is, in this continuation-passing programming style,* ***a function always "returns" its result by "sending" it to another function.*** (emphasis in original)

However, some Actor programming language constructs are not reducible to a continuation-passing style. For example, futures are not reducible to continuation-passing style.

The λ-calculus is capable of expressing some kinds of sequential and parallel control structures but, in general, *not* the concurrency expressed in the Actor Model.[53] On the other hand, the Actor Model is capable of expressing everything in the parallel λ-calculus and pure Logic Programs [Hewitt 2008f] and can be thousands of times faster for many Intelligent Applications [Hewitt 2012].

Early Actor Programming languages

Henry Lieberman, Dan Theriault, *et al.* developed Act1, an Actor programming language. Subsequently for his master's thesis, Dan Theriault developed Act2. These early proof of concept languages were rather inefficient and not suitable for applications. In his doctoral dissertation, Ken Kahn developed Ani, which he used to develop several animations. Bill Kornfeld developed the Ether programming language for the Scientific Community Metaphor in his doctoral dissertation. William Athas and Nanette Boden [1988] developed Cantor which is an Actor programming language for scientific computing. Jean-Pierre Briot [1988, 1999] developed means to extend Smalltalk 80 for Actor computations. Darrell Woelk [1995] at MCC developed an Actor programming language for InfoSleuth agents in Rosette.

Hewitt, Attardi, and Lieberman [1979] developed proposals for delegation in message passing. This gave rise to the so-called inheritance anomaly controversy in concurrent programming languages [Satoshi Matsuoka and Aki

[i] "*This is the synchronization primitive. It evaluates an expression uninterruptedly; i.e. no other process may run until the expression has returned a value.*"

Yonezawa 1993, Giuseppe Milicia and Vladimiro Sassone 2004]. ActorScript [Hewitt 2010] has proposal for addressing delegation issues.

Garbage Collection

Garbage collection (the automated reclamation of unused storage) was an important theme in the development of the Actor Model.

In his doctoral dissertation, Peter Bishop developed an algorithm for garbage collection in distributed systems. Each system kept lists of links of pointers to and from other systems. Cyclic structures were collected by incrementally migrating Actors (objects) onto other systems which had their addresses until a cyclic structure was entirely contained in a single system where the garbage collector could recover the storage.

Henry Baker developed an algorithm for real-time garbage collection is his doctoral dissertation. The fundamental idea was to interleave collection activity with construction activity so that there would not have to be long pauses while collection takes place.

Lieberman and Hewitt [1983] developed a real time garbage collection based on the lifetimes of Actors (Objects). The fundamental idea was to allocate Actors (objects) in generations so that only the latest generations would have to be examined during a garbage collection.

Cosmic Cube

The Cosmic Cube was developed by Chuck Seitz *et al.* at Caltech providing architectural support for Actor systems. A significant difference between the Cosmic Cube and most other parallel processors is that this multiple instruction multiple-data machine used message passing instead of shared variables for communication between concurrent processes. This computational model was reflected in the hardware structure and operating system, and also the explicit message passing communication seen by the programmer.

Communicating Sequential Processes

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 was received, execution of the computer was "interrupted" and special code called an interrupt handler was called to *put* the information in a 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 a single processor. 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. In contrast, the Actor Model does *not* take classical sequential processes as primitive and is *not* built on communicating sequential processes.

Dijkstra was certain that unbounded nondeterminism is impossible to implement. Hoare was convinced by Dikstra's argument. Consequently, the semantics of CSP specified bounded nondeterminism.

Consider the following program written in CSP [Hoare 1978]:

```
[X :: Z!stop( )                     ⓘ In process X, send Z a stop message
 ||                          ⓘ process X operates in parallel with process Y
 Y :: guard: boolean; guard := true;
                  ⓘ In process Y, initialize boolean variable guard to true and then
      *[guard→ Z!go( ); Z?guard]
      ⓘ while guard is true, send Z a go message and then input guard from Z
 ||                          ⓘ process Y operates in parallel with process Z
 Z :: n: integer; n:= 0;   ⓘ In process Z, initialize integer variable n to 0 and then
         continue: boolean; continue := true;
                          ⓘ initialize boolean variable continue to true and then
      *[                               ⓘ repeatedly either
         X?stop( ) → continue := false;
                 ⓘ input a stop message from X, set continue to false and then
                  Y!continue;          ⓘ send Y the value of continue
           []                          ⓘ or
         Y?go( )→ n := n+1;  ⓘ input go message from Y, increment n, then
                 Y!continue]]          ⓘ send Y the value of continue
```

According to Clinger [1981]:

> this program illustrates global nondeterminism, since the nondeterminism arises from incomplete specification of the timing of signals between the three processes X, Y, and Z. The repetitive guarded command in the definition of Z has two alternatives: either the stop message is accepted from X, in which case continue is set to false, or a go message is accepted from Y, in which case n is incremented and Y is sent the value of continue. If Z ever accepts the stop message from X, then X terminates. Accepting the stop causes continue to be set to false, so after Y sends its next go message, Y will receive false as the value of its guard and will terminate. When both X and Y have terminated, Z terminates because it no longer has live processes providing input.
>
> As the author of CSP points out, therefore, if the repetitive guarded command in the definition of Z were required to be fair, this program would have unbounded nondeterminism: it would be guaranteed to halt but there would be no bound on the final value of n. In actual fact, the repetitive guarded commands of CSP are not required to be fair, and so the program may not halt [Hoare 1978]. This fact may be confirmed by a tedious calculation using the semantics of CSP [Francez, Hoare, Lehmann, and de Roever 1979] or simply by noting that the semantics of CSP is based upon a conventional power domain and thus does not give rise to unbounded nondeterminism.

But Hoare knew that trouble was brewing because for several years, proponents of the Actor Model had been beating the drum for unbounded nondeterminism. To address this problem, he suggested that implementations of CSP should be as close as possible to unbounded nondeterminism! But his suggestion was difficult to achieve because of the nature of communication in CSP using nondeterministic select statements (from nondeterministic state machines, *e.g.*, [Dijkstra 1976]), which in the above program which takes the form

```
[X?stop( ) → ...
   []
 Y?go( ) → ...]
```

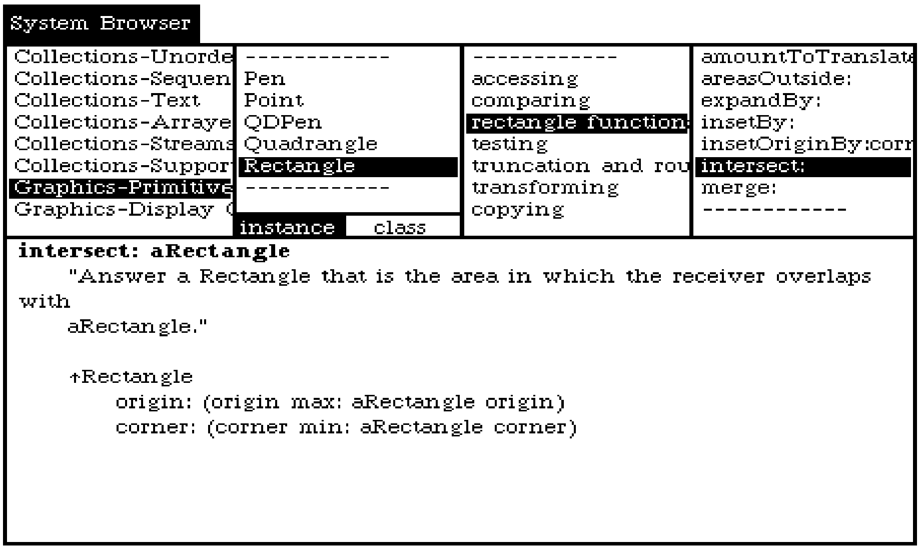
The structure of CSP is fundamentally at odds with guarantee of service.

Using the above semantics for CSP, it was impossible to formally prove that a server actually provides service to multiple clients (as had been done previously in the Actor Model). That's why the semantics of CSP were reversed from bounded non-determinism [Hoare CSP 1978] to unbounded non-determinism [CSP:1985]. However, bounded non-determinism was but a symptom of deeper underlying issues with nondeterministic transitions in communicating sequential processes (see [Knabe 1992]).

Smalltalk-80

Smalltalk-72 progressed to Smalltalk-80[Alan Kay, Dan Ingalls, Adele Goldberg, Ted Kaehler, Diana Merry, Scott Wallace, Peter Deutsch], which introduced the code browser designed and implemented by Larry Tessler[54] as an important innovation.

For example, the following diagram depicts a code-browser window:



In ActorScript, the above program fragment could be expressed as follows using strong types:

```
Actor Rectangle[origin:Point⊟, corner:Point⊟]
  uses rectangleFunction[ ]§
  intersect[aRectangle:Rectangle]:Rectangle ∴
     Rectangle[origin ⊟ max.[origin, aRectangle.⟦origin⟧],
                  corner ⊟ min.[corner, aRectangle.⟦corner⟧]]
```

Message passing model in Smalltalk-80 is closely tied to a particular machine model and programming language syntax that do not lend themselves to concurrency. Also, although the system is bootstrapped on itself, the behavior of language constructs are defined (like Lisp) by an interpreter instead of their response to **eval** messages. Furthermore, Smalltak-80, only has subclassing from a single superclass[55] and lacks formal interfaces for specifying types used in messages to communicate with other systems.

π-Calculus Actors

Robin Milner's initial published work on concurrency [Milner 1973] was notable in that it was not overtly based on sequential processes, although computation still required sequential execution (see below). His work differed from the previously developed Actor Model in the following ways:

- There are a fixed number of processes as opposed to the Actor Model which allows the number of Actors to vary dynamically
- The only quantities that can be passed in messages are integers and strings as opposed to the Actor Model which allows the addresses of Actors to be passed in messages
- The processes have a fixed topology as opposed to the Actor Model which allows varying topology
- Communication is synchronous as opposed to the Actor Model in which an unbounded time can elapse between sending and receiving a message.
- Unlike the Actor Model, there is no reception ordering and consequently there is only bounded nondeterminism. However, with bounded nondeterminism it is impossible to prove that a server guarantees service to its clients, *i.e.*, a client might starve.

Building on the Actor Model, Milner [1993] removed some of these restrictions in his work on the π-calculus:

> *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 ...*
>
> *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….*

However, some fundamental differences remain between the Actor Model and the π–calculus:

- The Actor Model is founded on physics whereas the π–calculus is founded on algebra.
- Semantics of the Actor Model is based on message orderings in the Computational Representation Theorem. Semantics of the π–calculus is based on structural congruence in various kinds of bi-simulations and equivalences.[56]

Process calculi (*e.g.* [Milner 1993; Cardelli and Gordon 1998]) are closely related to the Actor Model. There are similarities between the two approaches, but also many important differences (philosophical, mathematical and engineering):

- There is only one Actor Model (although it has numerous formal systems for design, analysis, verification, modeling, etc.) in contrast with a variety of species of process calculi.
- The Actor Model was inspired by the laws of physics and depends on them for its fundamental axioms in contrast with the process calculi being inspired by algebra [Milner 1993].
- Unlike the Actor Model, the sender is an intrinsic component of process calculi because they are defined in terms of reductions (as in the λ-calculus).
- Processes in the process calculi communicate by sending messages either through channels (synchronous or asynchronous), or via ambients (which can also be used to model channel-like communications [Cardelli and Gordon 1998]). In contrast, Actors communicate by sending messages to the addresses of other Actors (this style of communication can also be used to model channel-like communications using a two-phase commit protocol [Knabe 1992]).

There remains a Great Divide between process calculi and the Actor Model:

- *Process calculi:* algebraic equivalence, bi-simulation [Park 1980], *etc*.
- *Actor Model:* futures [Baker and Hewitt 1977], Swiss cheese, garbage collection, *etc.*

J–Machine

The J–Machine was developed by Bill Dally *et al.* at MIT providing architectural support suitable for Actors.

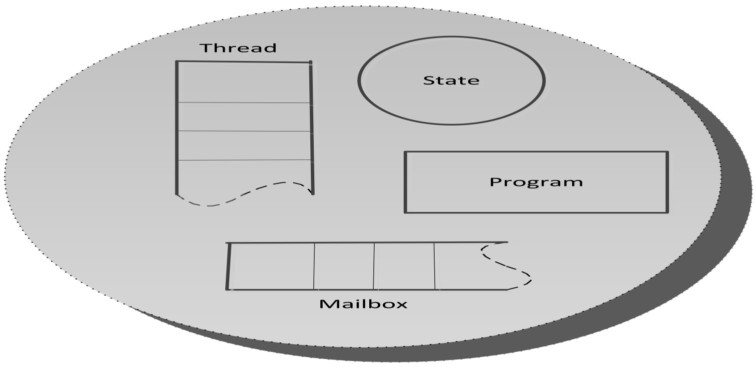
This included the following:

- Asynchronous messaging
- A uniform space of Actor addresses to which messages could be sent concurrently regardless of whether the recipient Actor was local or nonlocal
- A form of Actor pipelining

Concurrent Smalltalk (which can be modeled using Actors) was developed to program the J Machine.

“Fog Cutter” Actors

[Karmani and Agha 2011] promoted “*Fog Cutter*”[i] Actors each of which is required to have a mailbox, thread, state, and program diagrammed as follows:[57]



Process a message from the Mailbox using the Thread, then reset the Thread stack thereby completing the message-passing turn

Fog Cutter Actors are special cases in that the following restrictions hold:[ii]

- *Each Fog Cutter Actor has a ‘mailbox’.* But if everything that interacts is an Actor, then a mailbox must be an Actor and so in turn needs a mailbox which in turn … [Hewitt, Bishop, and Steiger 1973]. Of course, mailboxes having mailboxes is an infinite regress that has been humorously characterized by Erik Meijer as “down the rabbit hole.” [Hewitt, Meijer, and Szyperski 2012]
- *A Fog Cutter Actor ‘terminates’ when every Actor that it has created is ‘idle’ and there is no way to send it a message.* In practice, it is preferable to use garbage collection for Actors that are inaccessible. [Baker and Hewitt 1977]
- *Each Fog Cutter Actor executes a ‘loop’ using its own sequential ‘thread’ that begins with receiving a message followed by possibly creating more Actors, sending messages, updating its local state, and then looping back for the next message to complete a 'turn'.* In practice, it is preferable to provide “Swiss cheese” by which an Actor can concurrently process multiple messages without the limitation of a sequential thread loop. [Hewitt and Atkinson 1977, 1979; Atkinson 1980; Hewitt 2011]
- *A Fog Cutter Actor has a well-defined local ‘autonomous’ ‘state’ that can be updated*[58] *while processing a message.* However, because of indeterminacy an Actor may not be in a well-defined local independent state. For example, Actors might be entangled[59] with each other so that their actions are correlated. Also, large distributed Actors (*e.g.* www.dod.gov) do not have a well-defined state. It is usually preferable for an Actor not to change its local information while it is processing a message and instead specify how it will process the next message received (as in ActorScript [Hewitt 2011]).

---

[i] so dubbed by Kristen Nygaard (private communication).

[ii] “Fog Cutter” is in *italics*.

Fog Cutter Actors have been extremely useful for exploring issues about Actors including the following alternatives:

- **Reception order of messaging** instead of *Mailbox*
- **Activation order of messaging** instead of *Thread*
- **Behavior** instead of *State+Program*

However, Fog Cutter Actors are fundamentally lacking in generality because they lack the holes of Swiss cheese.[i]

In practice, the most common and effective way to explain Actors has been *operationally* using a suitable Actor programming language (*e.g.,* ActorScript [Hewitt 2012]) that specifies how Actors can be implemented along with an English explanation of the axioms for Actors (*e.g*., as presented in this paper).

Erlang Actors

Erlang Actors [Armstrong 2010] are broadly similar to Fog Cutter Actors:

1. Each Erlang Actor does not share memory addresses with other Erlang Actors.
2. An Erlang Actor can retrieve a message from its mailbox by selectively removing a message matching a particular pattern.

Erlang made important contributions by emphasizing the importance of the following:

- referential transparency
- failure handling

However, Erlang Actors have the following issues:

- Messaging in Erlang is not robust because a sent message will be dropped without warning if there is no Actor for the address.[ii]
- Erlang imposes high overhead in sending messages between Actors. For example, it imposes coordination overhead that messages sent between two Erlang Actors are delivered in the order they are sent.
- Implementations of Erlang do not make efficient use of many-core coherent architectures because messages between Erlang Actors must be blobs.[iii]
- Instead of using exception handling, until recently Erlang relied on process failure[iv] propagating between processes and their spawned processes.
- Instead of using garbage collection to recover storage and processing of unreachable Actors, each Erlang Actor must perform an internal termination or be killed externally.[60]
- Erlang does not have parameterized types, Actor aspects, interfaces or type discriminations.

[i] See section on Swiss cheese in this article.
[ii] Such silent failures are a bane of robust software engineering.
[iii] A blob is a data structure that cannot contain pointers.
[iv] based on an arbitrary time-out

Erlang Actors have been used in high-performance applications. For example, Ericsson uses Erlang in 3G mobile networks worldwide [Ekeroth and Hedstrờm 2000].

Squeak

Squeak [Ingalls, Kaehler, Maloney, Wallace, and Kay 1997] is a dialect of Smalltalk-80 with added mechanisms of islands, asynchronous messaging, players and costumes, language extensions, projects, and tile scripting. Its underlying object system is class-based, but the user interface is programmed as though it is prototype-based.

Orleans Actors

Orleans [Bykov, Geller, Kliot, Larus, Pandya, and Thelin 2010; Bernstein, Bykov, Geller, Kliot, and Thelin 2014] is a *distributed* implementation of Actors that transparently sends messages between Actors on different computers enabling greater scalability and reliability of practical applications.

Orleans is based on single-threaded Actor message invocations. An Actor processes a message using a thread from a thread pool. When the message has been processed, the thread can be returned to the thread pool.[61] That an Orleans Actor does not share memory with other Actors is enforced by doing a deep copy of messages if required. A globally unique identifier[62] is created for each Orleans Actor with a consequence that there is extra storage overhead that can be significant for a very small Orleans Actor.[63] A globally unique identifier can be used to send a message, which will, if necessary, create an activation[64] of an Orleans Actor in the memory of a process.[65]

Orleans has the following issues:

- Orleans allows the use of strings and long integers as globally unique identifiers in order to provide for perpetual Actors whose storage can only be collected using potentially unsafe means, which can result in a dangling globally unique identifier.
- A system design choice was made in Orleans not to use automated storage reclamation technology (garbage collection) to keep track of whether an Orleans Actor could have been forgotten by all applications and thus become inaccessible. Consequently, Orleans can have the following inefficiencies:
    - A short-lived Orleans Actor that has become inaccessible *does not have its storage in the process quickly recycled* resulting in a larger working set and decreased locality of reference.[66]
    - A long-lived Orleans Actor that has become inaccessible *does not ever have its storage recycled* [67] resulting in larger memory requirements.[68] However, collection of the storage of long-lived Actors is not so important in some applications because long-term memory has become relatively inexpensive.

An Orleans Actor ties up a thread while it is taking a turn to process a message regardless of the amount of time required, *e.g.*, time to make a system call. In this way, Orleans avoids timing races in the value of a variable of an Actor.[69]

A consequence of being single-threaded can be reduced performance of Orleans Actors as follows:

- lack of parallelism in processing a message
- lack of concurrency between processing a message and executing waiting method calls invoked by processing the message.[70]
- thread-switching overhead between sending and receiving a message to an Orleans Actor in the same process[71]

A waiting method call can be resolved using the **`await`**[72] construct as follows:

```
await anActor.aMethodName(…)
```
[73]

For example:

```
var anActor = aFactory.GetActor(aGloballyUniqueIdentifier);
try {…aUse(await anActor.aMethodName(…))…
       anotherUse(await anActor.anotherMethodName(…))…}
 catch …;
```
[74]

When reentrancy[75] is enabled, the method calls for *`aMethodName`* and *`anotherMethodName`* above are executed *after* the current message-processing turn:

- If completed successfully, the value of a waiting method call is supplied in a *new* turn at the point of method invocation, *e.g.*, the value of the method call for *`aMethodName`* of is supplied to *`aUse`*.
- If a waiting method call throws an exception, it is given to the exception handler in a new turn.

Orleans uses C# compiler "stack ripping" to use behind-the-scenes sequential turns to execute waiting method calls.

A message sent to an Orleans Actor must return a promise[76] Actor[77], which is a version of a future Actor. A promise Actor for a method call *`anActor.aMethodName`*`(…)` can be created using the following code:[78]

```
try {return Task.FromResult(await anActor.aMethodName(…));}
  catch (Exception anException)
    {return Task.FromException(anException);}
```
[79]

Note that a promise is *not* an Orleans Actor because it does not have a globally unique identifier.[80]

One of the motivations for the requirement that Orleans Actors must return promises when sent messages is to enable the **`await`** construct to *hide* promises so that clients of Orleans Actors do not have to deal with the return type `Task<T>` of each Orleans Actor method call for some application type `T`.

Orleans is an important step in furthering a goal of the Actor Model that application programmers need not be so concerned with low-level system details.[81] For example, in moving to the current version, Orleans reinforces the current trend of not exposing customer Actors[82] to application programmers.[83]

As a research project, Orleans had to make some complicated tradeoffs to implement more reliable distributed Actors. Implementing Actor systems that are both *robust* and *performant* is an extremely challenging research project that has taken place over many decades. More research remains to be done. However, Orleans has already been used in some high-performance applications including multi-player computer games, *e.g.*, Halo[Bykov 2013, Stenberg 2015].

JavaScript Actors

JavaScript Actors are broadly similar to Fog Cutter Actors.[84]

A *promise*[85] in JavaScript is a kind of future. JavaScript[86] will include asynchronous procedures as well as an **`await`**[87] construct that can be used to resolve promise Actors.

An asynchronous procedure *always*[i] returns a promise. For example, the following procedure computes a promise for the sum of two promises:

```
async function PromiseForSumOfPromises(aPromise, anotherPromise)
  {return (await aPromise) + await anotherPromise)};
```

A promise for an expression can be created by the procedure CreatePromise[88], which takes a thunk[89] for the expression as its argument. For example, suppose we have the following:[ii]

```
async function PromiseForSumOfTwoSlowCalls( )
  {const promise1 := CreatePromise(() => aSlowActor.do(10, 20));
   const promise2 := CreatePromise(() => aSlowActor.do(30, 40));
   return await PromiseForSumOfPromises(promise1, promise2)
  };
```

In an *asynchronous* procedure, **await** PromiseForSumOfTwoSlowCalls( ) is equivalent to the following in ActorScript:

(◎**Future** aSlowActor▪do[10, 20]) + ◎**Future** aSlowActor▪do[30,40]

---

[i] The use of asynchronous procedures can be contagious because a procedures using the return value of an asynchronous procedure needs to be asynchronous to use **await**.

[ii] The code is written in this way to emphasize that an asynchronous procedure *always* returns a promise.

To implement parallelism, JavaScript has workers.[90] Although multiple workers can reside in a process, they do not share memory addresses and consequently cannot efficiently communicate using many-core coherency. A worker communicates with other workers using blobs[i] in order to guarantee memory separation. Each worker acts as a *single-threaded, non-preemptive* time-sharing system for processing messages for Actors that reside in its memory.[91]

However, JavaScript workers have the following efficiency issues:[92]

1. There is no parallelism in processing messages for different Actors on a worker and the processing of a message by a slowly executing Actor *cannot* be preempted thereby bringing *all* [ii] other work on the worker to a *standstill*.[iii]
2. An Actor on a worker can directly send a message to an Actor on another worker only if the recipient has been transferred to the worker on which the sender resides.[93] An Actor can also indirectly send a blobbed message using a **MessageChannel**.
3. A very difficult efficiency issue is to decide how many Actors to put on each worker and which Actors to put on which worker.

JavaScript workers limit much of the modularity and efficiency available in coherent many-core processor architectures. Inherent inefficiencies and architectural deficiencies in JavaScript workers and HTML5 standards handicap[iv] browsers in their competition with apps.

Capabilities Systems

Capabilities were proposed in order to provide protection in operating systems [Dennis and van Horn 1966] by placing authority to take certain actions in special lists stored in protected memory of the operating system. Capabilities originated as part of the MIT Multics Project whereas Actors originated at the AI Lab, which developed Lisp machines with a tagged-memory architecture (instead of special lists stored in the operating system) that could be used to implement secure Actor addresses. Lisp machines were not commercially successful because the developing companies were under-capitalized and

[i] A blob is a data structure that cannot contain pointers. In the past, a more limited meaning called BLOB has been used as an acronym for Binary Large Object. In the Actor Model, an address (which is typed) can be used to send a message to an Actor. The model does not specify the physical representation of an address. So an address might be a (tagged) pointer. However, such pointers are not allowed in blobs.

[ii] including any queued promises

[iii] Issues of non-preemption motivated the invention of time-slicing [Bemer 1957] by which tasks are switched at the expiration of a timer.

[iv] due mainly to the legacy requirement not to break the Web. W3C and ECMA have done excellent work ameliorating the worst problems.

lacked an adequate software foundation. Unfortunately, tagged-memory architectures fell out of fashion subsequently causing enormous security problems in our current cyber systems.[94]

One of the motivations for developing the Actor Model in 1972 was that capabilities were awkward to use because their addresses were allocated in private memory of operating systems. Using tagged memory on Lisp Machines was a preferred implementation for Actors as opposed to using segmented memory on Multics. Also, the terms "capability" and "capabilities system" lacked axiomatizations and denotational semantics.

According to [Saltzer and Schroeder 1975]:

> In a computer system, a capability is an unforgeable ticket, which when presented can be taken as incontestable proof that the presenter is authorized to have access to the object named in the ticket.

In contrast:

> An Actor *address* is defined to be a shareable[i] digital token[95] that together with a type[ii] provides the ability to send a message[iii] to the address.[iv]

The following are some differences between Actor addresses and the Saltzer/Schroeder definition of capability:

- An address need not be unforgeable although it is typically unguessable[v].
- Unlike a capability[vi], an Actor address *per se* does not authorize anything. However, an Actor address together with a type enables a message to be sent to the address.[vii]
- A message sent to an address does not have to be honored. However, it is generally good practice for an Actor to respond with an exception if it dishonors a message.

---

[i] subject to type constraints

[ii] which is an Actor

[iii] which is an Actor

[iv] For example in the following, **HTTPS**["google.com"] is an Actor address that can be used as follows to send a **get** message to Google: **HTTPS**["google.com"]▪**get**[ ]

[v] For example, an Actor address 4 of type **Integer** is plainly not unguessable.

[vi] In a capability, designation and permissions are inextricably bound together.

[vii] The message might not actually be received for a variety of potential reasons. For example, because it is not properly received by an app for the intended recipient, because it is not properly received by a computer the for intended recipient, *etc*. It is good practice to throw an exception if a response is not received within some "reasonable" time.

According to [Levy 1985]:

> "Conceptually, a capability is a token, ticket, or key that gives the possessor permission to access an entity or object in a computer system. A capability is implemented as a data structure that contains two items of information: a unique object identifier and access rights."

The above notion of capability can be modeled as a very specialized proxy Actor that filters messages.

Historically, capabilities have been handicapped by *vagueness*, *over-specialization, awkwardness, not being integrated with types, being single-computer centric,* and for *not incorporating inconsistency-robust logic programs* which have persisted in various ways:[96]

- **Vagueness**: The [Saltzer and Schroeder 1975] definition above of a capability suffers from vagueness in specifying exactly what constitutes "access" to an object. In the Actor Model, a capability is often modeled as an Actor Address. An Actor type is needed for an address in order to perform marshalling, equivalence testing, and message sending and receiving.
- **Over-specialization**: Definitions of capabilities have been over-specialized. For example, the [Levy 1985] definition above of a capability suffers from the limitation of being over-specialized in that it specifies a particular data structure with two items.[i]
- **Awkwardness:** Capability systems have often been awkward to use. For example, it is awkward to program in a system that requires permissions to be kept in lists maintained in operating systems memory.[ii]
- **Single-computer centric:** Historically, capability systems have been single-computer centric. Keeping permissions in lists maintained in the operating system memory of a computer is an example.[97]
- **Not integrated with types:** Capabilities were not integrated with type systems. In the Actor Model, in order to use an Actor address to send a message, it is necessary to have a type for that address.
- **Not incorporating inconsistency-robust logic programs:**[98] Capabilities have not incorporated semantics of message passing using inconsistency-robust logic programs.

[i] More recently, over-specialization has become an aspect of being single-computer centric (see below).

[ii] More recently, awkwardness has become an aspect of being single-computer centric.

Capabilities were further developed in [Organick 1983; Levy 1984; Chander, Dean, and Mitchell 2001; Shapiro and Adams 2007; Woodruff, *et. al.* 2014; Watson, *et. al.* 2015]. Unfortunately, capabilities have continued to be awkward to use because their addresses were allocated in private memory of operating systems. [Kwon, *et. al*. 2014] is a tagged capability architecture that includes a special register to hold capabilities for addresses. Capabilities systems can be considered to be approaches to security making use of specified principles [Miller 2006] that must include the locality laws of the Actor Model [Baker and Hewitt 1977].

The vision that motivated creation and development of the Actor Model is now coming into fruition in the **Internet of Things** with the following aspects [Hewitt 2015/2016]:

- Systems must function robustly even though at any time a computer can become temporarily or permanently unavailable.
- Systems must function as robustly as possible even though connections between computers can be intermittent.

A citizen will share a great deal of sensitive personal information among their insulin pump, bedroom TV, cell phone, home router, and potentially even a brain implant (new DARPA project).[99] Consequently, security of sensitive information heavily relies on encryption.

One of the fundamental principles is to use unguessable addresses for Actors on remote computers.[100] In general, having an unguessable address and its type provides the computer that receives the unguessable address an opportunity to send a message to the address.[101]

The only ways that an Actor can acquire an unguessable address is to be given the information to compute it from a combination of the following:

1. Using addresses provided by creating other Actors
2. Using addresses received in messages

In practice, the information to compute an unguessable address must have come from the creator of the Actor for the unguessable address.[102] Consequently, using unguessable Actor addresses provides significant security in helping confine sensitive personal information in the IoT devices owned by a citizen.

The intent of a [Saltzer and Schroeder 1975] capability is to *guarantee authorization to have access*, whereas the intent of an Actor address is to provide *an opportunity to send messages*. An Actor address together with a type enables attempted sending a message to the address. However, having an Actor address together with a type doesn't guarantee anything about the ability to "use" any Actor. In IoT, a variety of circumstances (including failure to decrypt) might prevent a message from being received by its intended recipient and even if a message is received, a recipient might refuse to honor the message. For example, there may be no capability (bits) that can be specified ahead of time that mean that a withdrawal request to an account will be successful even if the account has funds to cover the withdrawal. When the account receives the withdrawal request it may require further evidence (perhaps from other parties).

Furthermore, an Actor address has functionality beyond that of a [Saltzer and Schroeder 1975] capability. For example, a type might use an Actor address in upcasting, downcasting, or casting to an interface of an Actor as in ActorScript (see below).

The following definition of "capability" for the *Internet of Things* aims for both *precision* and *practicality*:[103]

> A *capability* is defined to be an unguessable[104], shareable digital designation that indivisibly combines permission and ability to perform operations on an object[105] implemented on some system that has certain security properties, *e.g.*, those specified in the Actor Model for unguessable addresses.[106]

The following is an important difference between Capabilities and Actors:

- Capabilities are *prescriptive* specifying system properties which must hold.
- Actors are for *modeling* and *implementation*. Any digital computation can be *directly* modeled using Actors. ActorScript can *directly* efficiently implement any Actor system.[107] Of course, good engineering principles and practices should be strongly encouraged.

The interface type Account can be defined as follows:

```
Interface Account with availableBalance[ ] → Euro¶
                       deposit[Euro] → Void¶
                       withdraw[Euro] → Void▮
```

The following is an implementation SimpleAccount of Account:

```
Actor SimpleAccount[startingBalance:Euro]
  Locals myBalance ≔ startingBalance§
   // myBalance is an assignable variable initialized with startingBalance
  Implements Account using
    availableBalance[ ]:Euro ∴ myBalance¶
    deposit[anAmount:Euro]:Void ∴
       Void                                              // return Void
          afterward myBalance ≔ myBalance+anAmount¶
                      // the next message is processed with
                          // myBalance reflecting the deposit
     withdraw[anAmount:Euro]:Void ∴
       (amount > myBalance) �
          True ⦂ Throw Overdrawn[ ],
          False ⦂ Void                                   // return Void
                 afterward myBalance ≔ myBalance–anAmount ⍰§▮
               // the next message is processed with updated myBalance
```

The above implementation of SimpleAccount can be extended as follows to provide the ability to revoke some abilities to change an account.[108] For example, the AccountSupervisor implementation below implements both the Account and AccountRevoker interfaces as an extension of the implementation Account:

As illustrated below, a facet of an Actor can be expressed using "⍠" followed by the name of the interface for the facet.[109]

```
Actor AccountSupervisor[initialBalance:Euro]
            uses SimpleAccount[initialBalance]§
                        // uses SimpleAccount implementation 110
   Locals withdrawableIsRevoked ≔ False,
          depositableIsRevoked ≔ False§
   [[revoker]]:AccountRevoker ∴ ⍠AccountRevoker¶
                              // this Actor as AccountRevoker
   [[account]]:Account ∴ ⍠Account¶     // facet for this Actor as Account
   withdrawFee[anAmount:Euro]:Void →
               Void afterward myBalance ≔ myBalance−anAmount§
                     //  withdraw fee even if balance goes negative 111
                       //  myBalance is myBalance⊙SimpleAccount
   Partially reimplements Account using
    // (availableBalance[ ]→Euro) from SimpleAccount
    withdraw[anAmount:Euro]:Void ∴
      withdrawableIsRevoked �
        True ⦂ Throw Revoked[ ],
        False ⦂ ⍠SimpleAccount▪withdraw[anAmount] ⍰¶
                        // use withdraw of SimpleAccount
    deposit[anAmount:Euro]:Void ∴
       depositableIsRevoked �
          True ⦂ Throw Revoked[ ],
          False ⦂ ⍠SimpleAccount▪deposit[anAmount] ⍰§
   also implements AccountRevoker using
     revokeDepositable[ ]:Void ∴
       Void afterward depositableIsRevoked ≔ True¶
     revokeWithdrawable[ ]:Void ∴
       Void afterward withdrawableIsRevoked ≔ True§▮
```

For example, the following expression returns *negative* €3:

```
  (anAccountSupervisor ← AccountSupervisor▪[€3],
   anAccount ← anAccountSupervisor▪⟦account⟧,
   aRevoker ← anAccountSupervisor▪⟦revoker⟧,
   anAccount▪withdraw[€2]●                  // the balance is €1
   aRevoker▪revokeWithdrawable[ ]●
                                  // withdrawableIsRevoked is True
   Try anAccount▪withdraw[€5]         // try another withdraw
          catch� _ ⦂ Void ⍰●          // ignore the thrown exception112
                                          // myBalance remains €1
    anAccountSupervisor▪withdrawFee[€4]●
                     // €4 is withdrawn even though withdrawableIsRevoked
                                          // myBalance is negative €3
    anAccount▪availableBalance[ ]▮
```

One-way Messaging

The following is an implementation of an arithmetic logic unit that implements **jumpGreater** and **addJumpPositive** one-way messages:

```
Actor ArithmeticLogicUnit◁aType▷[]
  Implements ALU◁aType▷ using
    jumpGreater[x:aType, y:aType,
                    firstGreaterAddress:Address,
                    elseAddress:Address]:⊝ ∴
         InstructionUnit▫Execute[(x>y) �
                                       True ⦂ firstGreaterAddress,
                                       False ⦂ elseAddress ⍰]¶
     addJumpPositive[x:aType, y:aType, sumLocation:Location◁aType▷,
                          positiveAddress:Address, elseAddress:Address]:⊝ ∴
       (z ← (x+y),
        sumLocation �
           aVariableLocation:VariableLocation◁aType▷i ⦂
             (VariableLocation▪store[z]●
                             // continue after acknowledgement of store
              (z >0) � True ⦂ InstructionUnit▫execute[positiveAddress],
                       False ⦂ InstructionUnit▫execute[elseAddress] ⍰),
           aTemporaryLocation:TemporaryLocation◁aType▷ii ⦂
             (aTemporaryLocation▫write[z],
                           // continue concurrently with processing write
              (z >0) � True ⦂ InstructionUnit▫execute[positiveAddress],
                       False ⦂ InstructionUnit▫execute[elseAddress] ⍰) ⍰§▮
```

i VariableLocation◁aType▷ has store[aType] → Void▮

ii TemporaryLocation◁aType▷ has write[aType] → ⊝▮

Native types, e.g., JavaScript, JSON, Java, and HTML (HTTP)

Because Actor addresses are typed, almost any kind of addressed can be accommodated.

Object can be used to create JavaScript Objects. Also, **Function** can be used to bind the reserved identifier **This**. For example, consider the following ActorScript for creating a JavaScript object aRectangle (with length 3 and width 4) and then computing its area 12:

(aRectangle[i] ← **Object** {"length": 3, "width": 4]},
aFunction ← **Function** [ ]→ **This**⟦"length"⟧ ∗ **This**⟦"width"⟧,
Rectangle⟦"area"⟧ ≔ aFunction●
aRectangle⟦"area"⟧.[ ])▮

The setTimeout JavaScript object can be invoked with a callback as follows that logs the string "later" after a time out of 1000:

setTimeout⊙JavaScript.[1000,
Function [ ]→
console⊙JavaScript.["log"].["later"]]▮

HTML strings can be used to create Actor addresses. For example, the Wikipedia English homepage can be retrieved as follows:[113]

(HTTPS["en.wikpedia.org"]).get[ ]▮

JSON is a restricted version of **Object** that allows only Booleans, numbers, strings in objects and arrays.[ii]

Native types can also be used from Java. For example, if s:String⊙Java, then s.substring[3, 5][iii] is the substring of s from the 3rd to the 5th characters inclusive.

Java types can be referenced using **Refer**[iv], *e.g.*:

Refer java.math.BigInteger▮
Refer java.lang.Number▮

After the above, BigInteger.new["123"].instanceof[Number]▮ is equivalent to **True**▮.

---

[i] aRectangle is of type Object`JavaScript
[ii] *i.e.* the following JavaScript types are not included in JSON: Date, Error, Regular Expression, and Function.
[iii] substring is a method of the String type in Java
[iv] **Refer** is called **Import** in Java

Was the Actor Model premature?

The history of the Actor Model raises the question of whether it was premature.

Original definition of prematurity

As originally defined by [Stent 1972], "A discovery is premature if its implications cannot be connected by a series of simple logical steps to contemporary canonical or generally accepted knowledge." [Lövy 2002] glossed the phrase "series of simple logical steps" in Stent's definition as referring to the "*target community's ways of asking relevant questions, of producing experimental results, and of examining new evidence*." [Ghiselin 2002] argued that if a "*minority of scientists accept a discovery, or even pay serious attention to it, then the discovery is not altogether premature in the Stentian sense.*" In accord with Ghiselin's argument, the Actor Model was not premature. Indeed it enjoyed initial popularity and underwent steady development.

However, Stent in his original article also referred to a development as premature such that when it occurred contemporaries did not adopt it by consensus. This is what happened with the Actor Model partly for the following reasons:

- For over 30 years after the first publication of the Actor Model, widely deployed computer architectures developed in the direction of making a single sequential thread of execution run faster.
- For over 25 years after the first publication, there was no agreed standard by which software could communicate high level data structures across organizational boundaries.

Before its time?

According to [Gerson 2002], phenomena that lead people to talk about discoveries being before their time can be analyzed as follows:

> *We can see the phenomenon of 'before its time' as composed of two separate steps. The first takes place when a new discovery does not get tied to the conventional knowledge of its day and remains unconnected in the literature. The second step occurs when new events lead to the 'rediscovery' of the unconnected results in a changed context that enables or even facilitates its connection to the conventional knowledge of the rediscovering context*.

But circumstances have radically changed in the following ways:

- Progress on improving the speed of a single sequential thread has stalled for some time now. Increasing speed depends on effectively using many-core architectures.
- Better ways have been implemented that Actors can use to communicate messages between computers.
- Actors have been increasingly adopted by industry.

Consequently, by the criteria of Gerson, the Actor Model might be described by some as *before its time*.

According to [Zuckerman and Lederberg 1986], premature discoveries are those that were made but neglected. [Gerson 2002] argued,

> *But histories and sociological studies repeatedly show that we do not have a discovery until the scientific community accepts it as such and stops debating about it. Until then the proposed solution is in an intermediate state*."

By his argument, the Actor Model is a discovery but since its practical importance is not yet accepted by consensus, its practical importance is not yet a discovery.

Index

End Notes

[1] According to E.O. Wilson: "A colony of ants is more than just an aggregate of insects that are living together. ***One ant is no ant.***" [Suzuki 2014]

[2] The Actor model makes use of two fundamental orders on computational events [Baker and Hewitt 1977; Clinger 1981, Hewitt 2006]:

1. The *activation order* (⇝) is a fundamental order that models one event activating another (there is energy flow from an event to an event which it activates). The activation order is discrete:

$$\forall[e_1,e_2\in \text{Events}]\rightarrow \text{Finite}[\{e\in \text{Events} \mid e_1 \rightsquigarrow e \rightsquigarrow e_2\}]$$

There are two kinds of events involved in the activation order: reception and transmission. Reception events can activate transmission events and transmission events can activate reception events.

2. The *reception order* of an Actor x ($\underset{x}{\rightarrow}$) models the (total) order of events in which a message is received at x. The reception order of each x is discrete:

$$\forall[r_1,r_2\in \text{ReceptionEvents}_x]\rightarrow \text{Finite}[\{r\in \text{ReceptionEvents}_x \mid r_1 \underset{x}{\rightarrow} r \underset{x}{\rightarrow} r_2\}]$$

The *combined order* (denoted by ↷) is defined to be the transitive closure of the activation order and the reception orders of all Actors. So the following question arose in the early history of the Actor model: "*Is the combined order discrete?*" Discreteness of the combined order captures an important intuition about computation because it rules out counterintuitive computations in which an infinite number of computational events occur between two events (*à la* Zeno).

Hewitt conjectured that the discreteness of the activation order together with the discreteness of all reception orders implies that the combined order is discrete. Surprisingly [Clinger 1981; later generalized in Hewitt 2006] answered the question in the negative by giving a counterexample:

Any finite set of events is consistent (the activation order and all reception orders are discrete) and represents a potentially physically realizable situation. But there is an infinite set of sentences that is inconsistent with the discreteness of the combined order and does not represent a physically realizable situation.

The resolution of the problem is to take discreteness of the combined order as an axiom of the Actor model:[2]

$$\forall[e_1,e_2\in \text{Events}]\rightarrow \text{Finite}[\{e\in \text{Events} \mid e_1 \curvearrowright e \curvearrowright e_2\}]$$

Properties of concurrent computations can be proved using the above orderings [*e.g.* Bost, Mattern, and Tel 1995; Lamport 1978, 1979].

The above laws for Actor systems should be derivable from the laws of physics.

[3] for better or worse

[4] The receiver might be on another computer and in any the system can make use of threads, locks, location transparency, throttling, load distribution, persistence, automated storage reclamation, queues, cores, channels, ports, *etc.* as it sees fit.

Messages in the Actor model are generalizations of packets in Internet computing in that they need not be received in the order sent. Not implementing the order of delivery, 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 how 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.

[5] The amount of effort expended depends on circumstances.

[6] These laws can be enforced by a proposed extension of the X86 architecture that will support the following operating environments:

- CLR and extensions (Microsoft)
- JVM (Oracle, IBM, SAP)
- Dalvik (Google)

Many-core architecture has made the above extension necessary in order to provide the following:

- concurrent nonstop automated storage reclamation (garbage collection) and relocation to improve performance,
- prevention of memory corruption that otherwise results from programming languages like C and C++ using thousands of threads in a process,
- nonstop migration of Tutes (while they are in operation) within a computer and between distributed computers

[7] e.g., surgery, hormones, genes, anatomy, dress, pronouns, psychology, etc.

[8] The following is a interface for a customer that is used in request/response message passing for return type aType:

Interface Customer◁aType▷ with
  return[aType] ↦ ⊖¶
  throw[Exception] ↦ ⊖▮

[9] It is not possible to guarantee the consistency of information because consistency testing is recursively undecidable even in logics much weaker than first order logic. Because of this difficulty, it is impractical to test whether information is consistent.

[10] Consequently Describer makes use of direct inference in Direct Logic to reason more safely about inconsistent information because it omits the rules of classical logic that enable every proposition to be inferred from a single inconsistency.

[11] This section shares history with [Hewitt 2008f].

[12] *cf.* denotational semantics of the λ-calculus [Scott 1976]

[13] One solution is to develop a concurrent variant of the Lisp meta definition [McCarthy, Abrahams, Edwards, Hart, and Levin 1962] that was inspired by Turing's Universal Machine [Turing 1936]. If *exp* is a Lisp expression and *env* is an environment that assigns values to identifiers, then the procedure *Eval* with arguments *exp* and *env* evaluates *exp* using *env*. In the concurrent variant, **eval[***env***]** is a message that can be sent to *exp* to cause *exp* to be evaluated. Using such messages, modular meta definitions can be concisely expressed in the Actor model for universal concurrent programming languages (*e.g.* ActorScript [Hewitt 2010a]).

[14] However, they come with additional commitment. Inappropriate language constructs are difficult to leave behind.

[15] *E.g.* processes in Erlang [Armstrong 2007] and vats in the object-capability model[Miller 2006].

[16] Swiss cheese was called serializers in the literature.

[17] The programming language ML violates the Strong Typing requirement of having well-founded types because it has type **Any** which is called **Universal-Type**.

[18] The programming language ML violates the Strong Typing requirements for well-founded casting because it allows casting an Actor of any type to **Universal-Type**.

[19] In part, this section extends some material that was submitted to Wikipedia and [Hewitt 2008f].

[20] Turing [1936] stated:

> *the behavior of the computer at any moment is determined by the symbols which he* [the computer] *is observing, and his 'state of mind' at that moment*" and "*there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations*."

Gödel's conception of computation was formally the same as Turing but more reductionist in motivation:

> *There is a major difference between the historical contexts in which Turing and Gödel worked. Turing tackled the Entscheidungsproblem* [computational decidability of provability] *as an interesting mathematical problem worth solving; he was hardly aware of the fierce foundational debates. Gödel on the other hand, was passionately interested in the foundations of mathematics. Though not a student of Hilbert, his work was nonetheless deeply entrenched in the framework of Hilbert's finitistic program, whose main goal was to provide a meta-theoretic finitary proof of the consistency of a formal system "containing a certain amount of finitary number theory."* [Shagrir 2006]

[21] An example of the global state model is the Abstract State Machine (ASM) model [Blass, Gurevich, Rosenzweig, and Rossman 2007a, 2007b; Glausch and Reisig 2006].

[22] The λ-calculus can be viewed as the earliest message passing programming language [Hewitt, Bishop, and Steiger 1973] building on previous work.

For example, the λ-expression below implements a tree data structure when supplied with parameters for a leftSubTree and rightSubTree. When such a tree is given a parameter message “getLeft”, it returns leftSubTree and likewise when given the message “getRight" it returns rightSubTree:

```
λ[leftSubTree, rightSubTree]
  λ[message]  message � “getLeft” ⸰ leftSubTree
                        “getRight” ⸰ rightSubTree
```

[23] Allowing assignments to variables enabled sharing of the effects of updating shared data structures but did not provide for concurrency.

[24] [Petri 1962]

[25] Consequently in Simula-76 there was no required locality of operations unlike the laws for locality in the Actor mode [Baker and Hewitt 1977].

[26] The ideas in Simula-67 became widely known by the publication of [Dahl and Hoare 1972] at the same time that the Actor model was being invented to formalize concurrent computation using message passing [Hewitt, Bishop, and Steiger 1973].

[27] The development of Planner was inspired by the work of Karl Popper [1935, 1963], Frederic Fitch [1952], George Polya [1954], Allen Newell and Herbert Simon [1956], John McCarthy [1958, *et. al.* 1962], and Marvin Minsky [1968].

[28] This turned out later to have a surprising connection with Direct Logic. See the Two-Way Deduction Theorem below.

[29] According to [Kay 1993]:

> “This is a generalization of a stack frame, such as is used by the B5000, and very similar to what a good intermodule scheme would require in an operating system such as CAL-TSS—a lot of state for every transaction, but useful to think about.”

[30] Subsequent versions of the Smalltalk language largely followed the path of using the virtual methods of Simula-67 in the message passing structure of programs. However Smalltalk-72 made primitives such as integers, floating point numbers, etc. into objects. The authors of Simula-67 had considered making such primitives into objects but refrained largely for efficiency reasons. Java at first used the expedient of having both primitive and object versions of integers, floating point numbers, etc. The C# programming language (and later versions of Java, starting with Java 1.5) adopted the more elegant solution of using boxing and unboxing, a variant of which had been used earlier in some Lisp implementations.

[31] According to the Smalltalk-72 Instruction Manual [Goldberg and Kay 1976]:

> There is not one global message to which all message “fetches” (use of the Smalltalk symbols eyeball, ✓; colon, ⠅; and open colon, ⸰) refer; rather, messages form a hierarchy which we explain in the following

way-- suppose I just received a message; I read part of it and decide I should send my friend a message; I wait until my friend reads his message (the one I sent him, not the one I received); when he finishes reading his message, I return to reading my message. I can choose to let my friend read the rest of my message, but then I cannot get the message back to read it myself (note, however, that this can be done using the Smalltalk object *apply* which will be discussed later). I can also choose to include permission in my message to my friend to ask me to fetch some information from my message and to give that in information to him (accomplished by including **:** or ⦂ in the message to the friend). However, anything my friend fetches, I can no longer have.

In other words,

1) An object (let's call it the CALLER) can send a message to another object (the RECEIVER) by simply mentioning the RECEIVER's name followed by the message.
2) The action of message sending forms a stack of messages; the last message sent is put on the top.
3) Each attempt to receive information typically means looking at the message on the top of the stack.
4) The RECEIVER uses the eyeball, 👁 the colon, **:**, and the open colon, ⦂, to receive information from the message at the top of the stack.
5) When the RECEIVER completes his actions, the message at the top of the stack is removed and the ability to send and receive messages returns to the CALLER. The RECEIVER may return a value to be used by the CALLER.
6) This sequence of sending and receiving messages, viewed here as a process of stacking messages, means that each message on the stack has a CALLER (message sender) and RECEIVER (message receiver). Each time the RECEIVER is finished, his message is removed from the stack and the CALLER becomes the current RECEIVER. The now current RECEIVER can continue reading any information remaining in his message.
7) Initially, the RECEIVER is the first object in the message typed by the programmer, who is the CALLER.
8) If the RECEIVER's message contains an eyeball, 👁*;* colon, **:**, or open colon, ⦂, he can obtain further information from the CALLER's message. Any information successfully obtained by the RECEIVER is no longer available to the CALLER.
9) By calling on the object *apply,* the CALLER can give the RECEIVER the right to see all of the CALLER's remaining message. The CALLER can no longer get information that is read by the RECEIVER; he can, however, read anything that remains after the RECEIVER completes its actions.

10) There are two further special Smalltalk symbols useful in sending and receiving messages. One is the keyhole, ⍐, that lets the RECEIVER "peek" at the message. It is the same as the ꞉ except it does not remove the information from the message. The second symbol is the hash mark, #, placed in the message in order to send a reference to the next token rather than the token itself.

[32] The sender is an intrinsic component of communication in the following previous models of computation:

- *Petri Nets*: the input places of a transition are an intrinsic component of a computational step (transition).
- λ-c*alculus*: the expression being reduced is an intrinsic component of a computational step (reduction).
- *Simula-67*: the stack of the caller is an intrinsic component of a computation step (method invocation).
- *Smalltalk-72*: the invoking token stream is an intrinsic component of a computation step (message send).

[33] An Actor can have information about other Actors that it has received in a message about what it was like when the message was sent. See section of this paper on unbounded nondeterminism in ActorScript.

[34] Arbiters render meaningless the states in the Abstract State Machine (ASM) model [Blass, Gurevich, Rosenzweig, and Rossman 2007a, 2007b; Glausch and Reisig 2006].

[35] The logic gates require suitable thresholds and other parameters.

[36] Of course the same limitation applies to the Abstract State Machine (ASM) model [Blass, Gurevich, Rosenzweig, and Rossman 2007a, 2007b; Glausch and Reisig 2006]. In the presence of arbiters, the global states in ASM are mythical.

[37] Consider the following Nondeterministic Turing Machine:

Step 1: Next do either *Step 2* or *Step 3*.
Step 2: Next do *Step 1*.
Step 3: Halt.

It is possible that the above program does not halt. It is also possible that the above program halts. Note that above program is not equivalent to the one below in which it is not possible to halt:

Step 1: Next do *Step 1*.

[38] This result is very old. It was known by Dijkstra motivating his belief that it is impossible to implement unbounded nondeterminism. Also the result played a crucial role in the invention of the Actor Model in 1972.

[39] This proof does not apply to extensions of Nondeterministic Turing Machines that are provided with a new primitive instruction NoLargest which is defined to write a unbounded large number on the tape. Since executing NoLargest can write an unbounded amount of tape in a single instruction, executing it can take an unbounded time during which the machine cannot read input.

Also, the NoLargest primitive is of limited practical use. Consider a Nondeterministic Turing Machine with two input-only tapes that can be read nondeterministically and one standard working tape.

It is possible for the following program to copy both of its input tapes onto its working tape:

Step 1 : ***Either***

a) copy the next input from the 1st input tape onto the working tape and next do *Step 2,*

or

b) copy the next input from the 2nd input tape onto the working tape and next do *Step 3.*

Step 2 : Next do *Step 1*.

Step 3 : Next do *Step 1*.

It is also possible that the above program does not read any input from the 1st input tape (*cf*. [Knabe 1993]). Bounded nondeterminism was but a symptom of deeper underlying issues with Nondeterministic Turing Machines.

40 Consequently,

- The tree has an infinite path. ⇔ The tree is infinite. ⇔ It is possible that P does not halt.

  If it is possible that P does not halt, then it is possible that that the set of outputs with which P halts is infinite.

- The tree does not have an infinite path. ⇔ The tree is finite. ⇔ P always halts.

  If P always halts, then the tree is finite and the set of outputs with which P halts is finite.

41 [Kowalski 1988a]

42 A Logic Program is defined by the criteria that it must logically infer its computational steps.

43 A request to a shared resource might never receive service because it is possible that a nondeterministic choice will always be made to service another request instead.

44 [Nygaard 1986] Starting with Simula-67, which was not a pure Object programming language because for efficiency reasons numbers, strings, and arrays, were not Objects in the Class hierarchy.

45 [Knudsen and Madsen 1988]

46 According to [Nygaard 1986] (emphases in original):

> The term *object-oriented programming* is derived from the *object* concept in the Simula-67 programming language. ... Objects sharing a common structure are said to constitute a *class*, described in the program by a common *class description*.

[SIGPLAN 1986, Stein, Lieberman, and Ungar 1989] have discussions of object-oriented programming.

47 Examples of Object programming languages include Simula-67, Smalltalk-80, Java, C++, C#, and JavaScript. Recent Object languages support other

abstraction and code reuse mechanisms, such as traits, delegation, type classes, and so on, either in place of, or as well as inheritance.

[48] Every interface is a type and every type is an interface.

[49] [Kay 1998] wrote:

> *The big idea is "messaging" .... The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be. Think of the internet - to live, it (a) has to allow many different kinds of ideas and realizations that are beyond any single standard and (b) to allow varying degrees of safe interoperability between these ideas.*

[50] missing from initial versions of Scheme

[51] [Hewitt 1976, 1977].

[52] Notable members of this community included Bill Gosper, Richard Greenblatt, Jack Holloway, Tom Knight, Stuart Nelson, Peter Samson, Richard Stallman, *etc*. See [Levy 1984].

[53] The parallel λ-calculus includes the following limitations:

- Message reception order cannot be implemented.
- Actors that change cannot be implemented
- The parallel λ-calculus does not have exceptions.
- The fastest implementations of Intelligent Applications in the λ-lambda calculus can be thousands of times slower than Actor implementations.

[54] [Kay 1993]

[55] like Simula-67, which its runtime model is closely related

[56] Communication in the π -calculus takes the following form:

- *input:* u[x].P is a process that gets a message from a communication channel u before proceeding as P, binding the message received to the identifier x. In ActorScript [Hewitt 2010a], this can be modeled as follows:
  (x←u▪**get[ ]**● P)[56]
- *output:* ū[m].P is a process that puts a message m on communication channel u before proceeding as P. In ActorScript, this can be modeled as follows: (u▪**put[**x**]**● P)[56]

The above operations of the π-calculus can be implemented in Actor systems using a two-phase commit protocol [Knabe 1992; Reppy, Russo, and Xiao 2009]. The overhead of communication in the π–calculus presents difficulties to its use in practical applications.

According to [Berger 2003], Milner revealed

> *…secretly I realized that working in verification and automatic theorem proving…wasn't getting to the heart of computation theory…it was Dana Scott's work that was getting to the heart of computation and the meaning of computation.*

However, Milner continued his research on bi-simulation between systems and did not directly address the problem of developing mathematical denotations for general computations as in the Actor Model.

[57] *e.g.* as in Erlang [Armstrong 2010].

[58] *e.g.* using assignment commands

[59] a concept from (quantum) physics

[60] However, data structures *within* an Erlang Actor are garbage collected.

[61] which can be optimized by reusing the thread if another message is waiting

[62] a globally unique identifier can be a 128-bit guid, long integer, or a string.

[63] Also, a reference for an Orleans Actor can be created from a C# `anObjectAddress` using `aFactory.CreateObjectReference(anObjectAddress)`

[64] There can be optimizations for determinate message passing, *i.e.*, the same message always responds with the same result.

[65] Because of the ability to instantiate an Actor from its globally unique identifier, Orleans Actors are called "*virtual*" in their documentation. By analogy with virtual memory, the term "virtual" applied to an Orleans Actor would seem to imply that it would have to return to where it left. However, this terminology is misleading because an Actor can potentially migrate elsewhere and never come back.

Better terminology would be to say that an Orleans Actor is "*perpetual*."

[66] unless it is deleted by potentially unsafe means, which can result in a dangling globally unique identifier.

[67] after it has been unused for a while, its storage can be moved elsewhere outside the process in which it currently resides

[68] unless it is deleted by potentially unsafe means, which can result in a dangling globally unique identifier.

[69] ActorScript goes even further in this direction by enforcing that the value of a variable can change only when it is leaving the cheese or before/after an internal delegated operation.

[70] However, after the message is finished processing, sometimes waiting method calls it invoked can be processed concurrently if they are independent.

[71] provided that the Actor is not contended

[72] [Microsoft 2013]

[73] ActorScript uses `◎aFuture` to resolve `aFuture`

[74] In ActorScript. the program is:

```
Try …aUse(◎anActor.aMethodName(…))…
        anotherUse(◎anActor.anotherMethodName(…))…
    catch …
```

[75] reentrancy allows execution of waiting method calls to be freely interleaved

[76] [Liskov and Shira 1988; Miller, Tribble, and Shapiro 2005]

[77] Orleans uses `Task<aType>` for the type of a promise which corresponds to the type `Future◁aType▷` in ActorScript.

[78] ActorScript uses **Future** `anExpression` to create a future for `anExpression`

[79] There is an inefficiency in the above code in that the method call returns a promise that is taken apart and then an equivalent promise is created to be returned.

[80] It would be impractical for promises to be Orleans Actors because

- they are created as the return value of *every* Orleans Actor method call
- the storage of inaccessible Orleans Actors is *not* recovered, *e.g.*, using garbage collection

[81] *e.g*. threads, throttling, load distribution, cores, persistence, automated storage reclamation, locks, location transparency, channels, ports, *etc.*

[82] for requests, *e.g*., method calls. Customers are sometimes called continuations in the literature although continuations often cannot handle exceptions.

[83] However, Orleans does still surfaces continuations using lower level primitives.

[84] [ECMA 2014]

[85] Promise Actors were sometimes called "futures" in the beginning [Russell 2013, Yoshino 2013].

[86] [Barton 2014]

[87] somewhat analogous the **await** construct in C# [Microsoft 2013]

[88]
```
function CreatePromise(thunkForExpression)
    {return Promise.resolve(true)
                    .then((aValueToDiscard) =>
                             thunkForExpression())};
```

[89] A thunk is an intermediary procedure for assistance in carrying out a task [Church 1941, Ingerman, 1961].

[90] A JavaScript worker can be modeled as a Tute.

[91] Of course, at a different level of abstraction, workers can also be modeled as Actors that communicate with other workers.

[92] roughly in order of decreasing importance

[93] JavaScript has transferable Actors (which include the types **ArrayBuffer, MessagePort**. and **WorkerCanvas**) by various mechanisms. According to [Miller 2016] with regard to the ArrayBuffer detachment mechanism:

> "It is an ownership transfer of exclusive access to the underlying data area in memory. When one worker does a postMessage of an ArrayBuffer to another, the sender loses access to the area of memory in which those bytes are stored. The receiver received a new ArrayBuffer object born providing access to the data in that same area of memory."

[93] due mainly to the legacy requirement not to break the Web. Under difficult circumstances, W3C and ECMA have worked to clean-up and make extensions without breaking the Web.

[94] In order to address these security problems, tagged-memory architectures need to be created as extensions of current ARM and Intel architectures.

[95] Not requiring that an Actor address is always required to be unguessable can allow for more efficient implementations. For example, suppose that amount1 and amount2 are both of type **Euro**. It might be that case that amount1 and amount2 are unboxed, *i.e.*, the respective amounts are encoded in their addresses.

[96] Capabilities were critiqued in [Bobert 1984; Rajunas 1989; Miller, Yee and Shapiro 2003] concerning the following issues:

- *revocability*: An Actor does not have to honor the message that it receives. Using proxies for Actors also enables revocability because messages are forwarded and so a proxy can revoke.
- *multi-level security*: Actors, *per se*, do *not* have levels of security although various security schemes can be implemented, which may require using membranes [Donnelley 1976, Hewitt 1980].
- *delegation*: Actors directly support delegation by passing addresses of Actors in messages. However, a receiver must have appropriate types in order to send messages to addresses that it has received.
- *confinement*: Actor can use encryption to help enforce secrecy of information. For example, a computer might accept communications to Actors that it hosts only if the communication is encrypted by certain other computers.

[97] [Donnelley 1975] extended a single-computer capability system across multiple computers using ports to communicate protected capabilities. However, the extension does not work well for IoT in which devices are provided by many different manufactures with different operating systems

[98] This handicap is of more recent vintage that the ones above.

[99] Without warning, any of the above may fail permanently and have to be replaced. While the replacement is happening, life must continue as smoothly as possible.

[100] both of the following are needed to securely and efficiently implement Actor addresses [Hewitt 1980, Miller 2006]:

1. tagged pointers in an address space
2. unguessable addresses sent to other computers

[101] The message might not actually be received for a variety of potential reasons. For example, because it is not properly received by an app the for intended recipient, because it is not properly received by a computer for the intended recipient, *etc*. It is good practice to throw an exception if a response is not received within some "reasonable" time.

[102] Bits cannot be converted into an Actor address of arbitrary type, i.e., an Actor cannot convert an address of type **BitString** into an address of type **Account**.

[103] The following are examples of capabilities:

- Waterken [Close 2008]: an Actor address of type **WebKey**
- Zebra Copy [Karp and Li 2007]: an Actor address together with additional information that includes a list of allowed message types

104 In the Internet of Things, a designation of an object on a remote computer is unguessable but technically not unforgeable because it makes use of encryption for security. However, it is possible to use a capability that is technically unforgeable even though it is based on an unguessable (but forgeable) designation. However, it could be misleading to simply say that a capability is unforgeable when it is based on an unguessable designation.

105 *i.e.*, Java, Hypertext Transfer Protocol, C++, *etc*. The object might reside on a computer that is remote from on the one on which the operation is being invoked.

106 The use of types in the Actor Model has some similarities and differences with Split Capabilities[Karp, Gupta, Rozas, and Banerji 2003]. In order to send a message to an Actor, both and address and a type are required. A difference is that that this division is not the same as in Split Capabilities because a type is an Actor in its own right and not restricted to being a list of access rights. For example, if anAccount is of type **Account** then, anAccount.**deposit**[$5] is equivalent to the following:

Account.**send**[anAccount, **deposit**[$5]]

107 By default, ActorScript systems adhere to the prescriptions of capabilities and furthermore have stronger security properties as well, *e.g.*, making using of type encryption and not allowing insecure casting.

108 The ability to extend implementation is important because it helps to avoid code duplication.

109 *cf*. [Crahen 2002, Amborn 2004, Miller, et. al. 2011]

110 note the absence of "." in the **implementation** subexpression

111 equivalent to the following:

myBalance⊙SimpleAccount ≔
myBalance⊙SimpleAccount – anAmount

112 ignoring exceptions in this way is *not* a good practice

113 equivalently (HTTPS["en.wikpedia.org"])⟦ ⟧