Strengthening Process Calculi

Mingsheng Ying: Topology in process calculus: approximate correctness and infinite evolution of concurrent programs.

Since process calculi come up now and then, I borrowed this book from the library and tried to read it. Cannot claim to have grokked it, but I was excited reading through the example applications toward the end of it. I (think I) am hoping this kind of work can find its way into the main-stream somehow, some day.

> *Communication and concurrency are essential in understanding complex dynamic systems, and there have been many theories to deal with them such as Petri nets, CSP and ACP. Among them, process calculus is one of the most important and mathematically developed models of communication and concurrency. Various behavior equivalences between agents, such as (strong and weak) bisimilarity, observation congruence, trace equivalence, testing equivalence and failure equivalence, are central notions in process calculus.*
>
> *In the real applications of process calculus, specification and implementation are described as two agents, correctness of programs is treated as a certain behavior equivalence between specification and implementation, and then the proof of correctness of programs is a task to establish some behavior equivalence between them.*
>
> *The goal of this book is to provide some suitable and useful concepts and tools for the understanding and analysis of approximate correctness of programs in concurrent systems. Throughout this book the focus is on the framework of process calculus, and the main idea is to construct some natural and reasonable topological structures which can reveal suitably a mechanism of approximate computation in process calculus and to work out various relationships among processes which are compatible with these topological structures.*

By raould at 2015-06-17 21:28 | LtU Forum | previous forum topic | next forum topic | other blogs | 10416 reads

Comment viewing options

Threaded list - expanded | Date - oldest first | 200 comments per page | Save settings

Select your preferred way to display the comments and click "Save settings" to activate your changes.

How do you stop an actor being shared?

I really don't get this. How do you stop an actor being shared. When I compile a code module I have no idea which threads are going to call my actor.

It seems actors break separate compilation, which makes them unusable for any general programming language where compiling large programs is a must.

By Keean Schupke at Mon, 2015-06-29 17:18 | login or register to post comments

re how do you stop an actor being shared?

Lexical scope.

By Thomas Lord at Mon, 2015-06-29 17:24 | login or register to post comments

Pure and Impure Actors

If I have a maths library of common maths functions like factorial, how would I get local, lexically scoped copies of these actors?

If the actors are pure, I don't think we actually need local lexically scoped copies, as they have no state.

We only need local copies of stateful actors. How can I tell the if an actor is stateful to determine if I need a local lexically scoped copy?

But even with lexically scoped actors, it does not seem trivial to determine if multiple threads can simultaneously call the actor.

Edit: To clarify, it seems we need a type system that tracks effects, or has monads like Haskell, so that whether an actor is pure or impure can be tracked in the type system. A pure actor can be used direct from the definition without instantiation. An impure actor needs to be initialised (in a lexical scope thus controlling sharing to some degree). I am still not clear how decisions to parallelise (requiring mutex locks) or serialise actors would be taken.

By Keean Schupke at Mon, 2015-06-29 17:38 | login or register to post comments

It's up to recipient how it processes a message

It's up to recipient how it processes a message. If the recipient claims to be cacheable (i.e. same response for same message) then the infrastructure can cache responses.

Of course, no matter what the implementation mechanism, a shared financial account must deal with contention in massively concurrent systems.

By Hewitt at Mon, 2015-06-29 20:11 | login or register to post comments

Determinate

Caching responses is going to be slower than just running the actor for small actors. Especially as the cache will need mutex locks for updating the cache when used from multiple threads. The pure version can be shared between threads without needing any locks.

It seems to me we need to be able to determine if an actor is determinate in the type system (Monads do this), we need to be able to track side effects (So an effects system is needed in the type system too). We also need to be able to represent thread access in the type system so that we can implement:

	single thread	multi thread
pure	no mutex	no mutex
stateful	no mutex	mutex

If we can track the information in the above table in the type system we should be able to keep the use of mutexes to where there really are multiple threads accessing a stateful actor.

By Keean Schupke at Mon, 2015-06-29 20:36 | login or register to post comments

I could design one, but I'm not enough convinced

that it's worth it to invest the time.

For instance on a simple symmetric multiprocessing system, I'd do my usual vm tricks:
1) one system thread per core or per hardware hyperthread, green threading within each system thread
a) this allows nearly free locks within each system thread (but not touching the other threads)
b) this allows synchronization between ALL threads at a cost only commensurate with the number of hardware cores, since context switches only happen at safe points
c) this sort of system is almost necessary for high quality, parallel garbage collection anyway, and is used (I think) by both java and .net

2)actors that aren't pure (and therefore need to be synchronized) are owned by a given core/system thread
a) they only need one set of i/o queues per core
b) since writes to a given queue are always by the same core, no synchronization primitives are needed on processors like Intel. On processors like current generation Arm, then there will need to be some cache flushing per message that communicates with a different core. Arm is a worst case where there needs to be a synchronization instruction on load if there is even a possibility that something was written by a different core so the best you could do on Arm is to aggregate a bunch of unrelated message checks with one synchonization instruction.
c) I originally forgot to add: a call on an actor from the processor that owns it is just a function call. And on a processor like Intel a check for which thread owns it can be unsynchronized. If you want a check like that to be unsynchronized on Arm, then you'd need to sync all of the processors whenever an actor switched owning processors.

Note: Arm 8 promises to be the first implementation of C++ memory model in hardware. At this rate it will never be released, so by Arm I mean 7 or under.

By Josh Scholar at Wed, 2015-07-01 05:08 | login or register to post comments

Purity of actors, and threads

I am reasonably sure an effect system modelled by parametric monads and row polymorphism can track the purity and impurity of actors in a way that does not limit parallelism.

The interesting question for me is how to track the number of threads that may send a message to an actor. The number of threads is something like the number of messages in flight at a time. If all the actors wait for a response after sending a single message, and we start the program by injecting a single message, then there would only ever be a single thread needed. If an actor sends messages in parallel to the same recipient they have to be sequentialised. So the only time we introduce new threads is when an actor sends multiple messages in parallel to different receivers, and the only time we need a lock is when these different receivers, or actors they message, message the same actor.

As we have to know the "address" of an actor to message it, the above condition becomes, an actor needs a mutex if its address is sent to more than one actor in parallel, which each then use the address.

It should be possible to capture this information in the type system using monadic effects. If there is an effect for sending a message to address X, then this tracks sequential messaging within a thread. The tricky part is we need the effects to combine differently when messaging multiple actors in parallel that may message the same actor.

By Keean Schupke at Wed, 2015-07-01 06:11 | login or register to post comments

I'm a disbeliever

I don't even know what that stuff you wrote meant, but at the bottom, processors (some of them) have read and/or write queues on

Browse archives
« June 2017

Su	Mo	Tu	We	Th	Fr	Sa
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

Active forum topics
- Free links to all (or practically all) recent SIGPLAN papers
- Refining Structured Type System
- Finding Solutions vs. Verifying Solutions
- A refutation of Gödel's first incompleteness theorem
- Any thoughts on WanaDecrypt0r?

more

New forum topics
- Free links to all (or practically all) recent SIGPLAN papers
- SW verification continues
- Finding Solutions vs. Verifying Solutions
- the type of eval in Shen
- Any thoughts on WanaDecrypt0r?

more

Recent comments
- More than COBOL 15 hours 45 min ago
- Many articles inaccessible 5 days 3 hours ago
- The ACM DL remains a second choice for me 5 days 6 hours ago
- I can't read your example 1 week 1 day ago
- Subtyping 1 week 2 days ago
- Yes I agree 1 week 2 days ago
- But the "everyday language" 1 week 2 days ago
- I should probably be more 1 week 3 days ago
- Tx for support, 1 week 4 days ago
- Retract, new try 1 week 4 days ago

memory - these are hardware level, not instruction level.

If multiple processors are going to communicate, they have to have a design that works with those queues, and those designs can either be naive ones with lots of slow instructions that make those queues flush and wait for them, OR they can be designed as I posted to finesse the actual behavior of those queues and match it to needs. The design I mentioned trades memory for locks and limits the amount of memory needed to the number of processors, at a further cost of changing the threading model. But it also means that the system can optimize actor methods by making calls from SOME THREADS cheap.

I don't see how your gobbletygook changes that, underneath all your abstractions, you still need a mechanism for synchronizing memory between processors, especially on processors like Arm that make no guarantees and essentially can have NO kinds of safe memory sharing without synchronization instructions. Although it's on processors that have some guarantees where you can actually finesse the code to be most efficient. But aggregating synchronization on Arm is a big win over naive too.

By Josh Scholar at Wed, 2015-07-01 06:32 | login or register to post comments

Ah maybe you didn't disagree. I was only talking about 1 case

I was only talking about the case of impure actors.

So I was assuming that you had already sorted the pure ones from the impure ones.

By Josh Scholar at Wed, 2015-07-01 06:42 | login or register to post comments

Multiple threads

I am assuming we have already sorted pure from impure actors (using some kind of monadic effects, and tracking this in the type system). I was discussing tracking whether an impure actor is ever passed a message from more than one thread. So even if an actor is impure, if all messages sent to it come from the same thread it does not need a mutex lock.

By Keean Schupke at Wed, 2015-07-01 07:32 | login or register to post comments

My idea is more permissive

My idea makes SOME calls a lot faster but others even slower if latency counts - though I can imagine a work around.

My design, rather than depending on mutexes, depends on queues being processed in another thread, when they're noticed, which should make message latency very high - but since it doesn't rely on synchronized instructions it could make a system with enough parallel work have better throughput.

A work around could free unused actors from their thread if they go unused for long enough - in that case calling one costs as much as a mutex, but only if they've been freed.

You wrote:
"As we have to know the 'address' of an actor to message it, the above condition becomes, an actor needs a mutex if its address is sent to more than one actor in parallel, which each then use the address."

First of all, it's work for the programmer to convince the compiler that only one thread has the address of an actor at a time. More work than it's probably worth.

Second of all, I was saying that you can optimize for calls from a group of thread WITHOUT there being limits on who can call that actor, it's just that calls from the wrong threads will wait until they're handled.

Note, that *switching* ownership between threads isn't cheap unless you can prove to the program that there is no possibility of messaging from multiple threads so see objection 1, that proving this to the compiler would be a lot of work.

*****

Consider the case of optimizing a system like the one I described - that would be a matter of trying to untangle the web of calls to balance the load while reducing the number of messages that have to wait.

Also if the speed of communication between threads depends on those threads sharing a processor, then scheduling is more constrained than is traditional.

Perhaps one just uses hints from the programmer. I'm a fan of run-time profiling and analysis. Or whatever you call it when you do some runs for analysis and then use the data from those runs to recompile with better optimization.

By Josh Scholar at Wed, 2015-07-01 08:07 | login or register to post comments

Maybe the solution is:

Even though an actor design is one way to attempt to make parallel algorithms reliable, it can't make communication between different processors fast so you:

1) design a program with large chunks that are expected to run on a single processor. Code is all labeled by which chunk it's in.

2) the system is free to assign these large chunks among processors or move them, but as a whole.

3) there can be connections of different strength between the chunks so that more strongly connnected chunks tend to be on a single processor, or weakly connected ones always go on separate processors for efficiency

By Josh Scholar at Wed, 2015-07-01 08:24 | login or register to post comments

Actors all the way down

I agree that green threads running on each core, with locks makes sense. There may also be efficiencies that can be gained from cores sharing a cache (mutexes can be in cache, rather than in main memory).

I think that makes sense if you are modelling message passing using actors written in some other language.

I was trying to imagine a way of compiling that could make actors all the way down comparable in efficiency to an imperative languages, hence why it is important to know if actors are only used sequentially.

By Keean Schupke at Wed, 2015-07-01 09:19 | login or register to post comments

####### I'm curious what it would be like

to build something like Self or Smalltalk or some such on message passing concurrency.

Functions aren't actors in that they return values back and whatever invoked them waits for an answer... c2.com has a page on actor languages that claims that scheme came from an attempt to understand the actor model followed by a paragraph that I suspect was written by Hewitt:
Sussman and Steele arguably got continuations wrong; continuations in the actor model do not have the security problems that caused them to be omitted from DoubleYouSeven. In particular, Scheme continuations can return more than once to code that is unprepared for this. In the actor model, OTOH, an actor can be written to respond a continuation more than once, but that would have to be explicitly programmed. Also note that actor continuations are a derived concept, whereas in implementations of Scheme they have to be supported as a primitive.
(here http://c2.com/cgi/wiki?ActorLanguages)

It's not clear to me how that would work, but typing a function for continuations that capture it is interesting.

Note, I just followed the link on that page to Actor Prolog, which while that looks a very nice discussion, it's using a totally unrelated meaning for the word "actor" and so doesn't belong on that page.

By Josh Scholar at Wed, 2015-07-01 11:39 | login or register to post comments

######## Functions aren't actors in

> Functions aren't actors in that they return values back and whatever invoked them waits for an answer.

This isn't difficult to support by binding an invocation result to a future. The E language distinguishes method 'send' from 'call', where 'send' does this binding implicitly.

By naasking at Wed, 2015-07-01 12:32 | login or register to post comments

######### Some actors are funcitons

If you send the resumption or continuation with the message, and then stop, then the result is the same as a function call.

I would have thought it would be good to have a function-call like syntax for this, because languages like JavaScript that have event models are awkward to program.

By Keean Schupke at Wed, 2015-07-01 13:10 | login or register to post comments

########## There is nothing as unreadable as CPS

I hope no one is expected to use that without syntactic sugar.

By Josh Scholar at Wed, 2015-07-01 13:14 | login or register to post comments

######### Procedures are Actors with message passing, e.g., Factorial.[3]

Procedures are Actors with message passing, e.g., Factorial.[3]

By Hewitt at Wed, 2015-07-01 13:16 | login or register to post comments

########## Continuations

Is that supposed to work by sending a message containing "3" and the continuation to call with the result to the "factorial-actor"?

By Keean Schupke at Wed, 2015-07-01 14:58 | login or register to post comments

########### A request conceptually has a customer Actor

A request conceptually have a customer Actor. A customer Actor can be sent a *return* or a *throw* communication.

By Hewitt at Wed, 2015-07-01 15:05 | login or register to post comments

Changed?

Has that changed from the earlier descriptions with continuations? When a actor sends a request to a customer, does it wait for the reply/return?

By Keean Schupke at Wed, 2015-07-01 15:46 | login or register to post comments

Customer Actors are ancient; continuations in cheese are new

Customer Actors are ancient; continuations in cheese are new.

By Hewitt at Wed, 2015-07-01 20:36 | login or register to post comments

Customer vs Cheese

Why did you explain using customer actors in the post before last, only to tell me later that is out of date?

So what would a function call look like using the newer model?

By Keean Schupke at Thu, 2015-07-02 06:03 | login or register to post comments

keen

customer is "not out of date".

By Thomas Lord at Thu, 2015-07-02 06:10 | login or register to post comments

So there are two models?

Why would you have two models for message passing when one subsumes the other? What's wrong with sending the continuation with the message? If forking a new thread is explicit, then you can capture both options, you either send a message with the continuation without forking (works like a function call) or send a message with a continuation in a new thread.

By Keean Schupke at Thu, 2015-07-02 06:23 | login or register to post comments

Addres of customer conceptually sent with request

Conceptually, the address of a customer Actor is sent with a request.

Continuations are a concept used in ActorScript to structure execution in cheese.

By Hewitt at Thu, 2015-07-02 07:15 | login or register to post comments

Confused

So when I write an actor program like:

```
print[f[3]]
```

What determines if I wait for the reply to the message '3' sent to 'f'?

By Keean Schupke at Thu, 2015-07-02 07:28 | login or register to post comments

Type, address, and recipient determine how request is processed

Type, address, and recipient Actor determine how a request is processed in ActorScript.

By Hewitt at Thu, 2015-07-02 07:42 | login or register to post comments

Code reuse.

If you could send a continuation with any message, and either send the message in the current thread or in a new-thread, then you could re-use other actors more, as the same message handler could be used synchronously (like a procedure call) or asynchronously (like starting a new thread).

If the receiver determines the message semantics, then common functionality needs to be implemented twice for the synchronised (procedure call) and asynchronous (message passing) communication methods.

By Keean Schupke at Thu, 2015-07-02 11:05 | login or register to post comments

Threads are a terrible conceptual model of concurrency

Threads are a terrible conceptual model of concurrency causing much confusion.

Of course, low level implementation mechanisms, e.g., registers, coherent memory, caches, stacks, threads, *etc.* can be very useful.

By Hewitt at Thu, 2015-07-02 11:30 | login or register to post comments

But we have to have them?

Maybe, but how else can you implement concurrency? A thread is defined by having the same memory and resources (so we don't have to reprogram the MMU all the time), but a different stack for local variables.

In terms of implementation I don't think there are any alternatives, better or otherwise.

In terms of the model, all I am suggesting is that the sender determines whether to wait for a response, rather than the receiver. This seems better for code reuse as the same actor message-hander can be used in both cases.

By Keean Schupke at Thu, 2015-07-02 11:53 | login or register to post comments

efficiency and pervasive inconsistency (re https)

[Pretty much what Hewitt already said in reply but something more at the end.]

It looks to me that, for example, the lexically scoped and no-global-state nature of actor programs means that a lot of code that looks like a transcription of something you'd write in C++ is easy to recognize as being easily compiled using similar techiques.

If your code on the other hand makes use of concurrency then I guess compilation starts to have dealing with scheduling. Probably won't be using a simplistic stack discipline.

If your code has or might have sharing of actors among concurrently running actors, only then do message sends get overheads associated with arbitration of sends.

If you have loosely coupled components (e.g. parts of a system that can be separately updated on the fly) at least you are in the familiar territory where it pays to "be strict in what you trasnmit, tolerant in what you receive".

> What happens if this results in the sender and receiver having different types for the same message. Surely any assumption the sender knows the correct types to send will result in a security hole, and so there needs to be a protocol where the types are encoded into the message and the receiver can reject the message if they don't match what it is expecting.

More generally: consider a system that partitions into two set of communicating actors, A and B. And say their is either unilateral or bilateral mistrust. A has to consider that B will malfunction and spew garbage results and/or vice versa.

One suggestive possibility, if Direct Logic is regarded as elements of a logic programming language, is for A to treat the messages from B as containing assertions B makes in what A can think of as "Theory B". Every message B sends is something like a proposition that B says is true in "Theory B".

In that way A winds up with a partial model of Theory B without having to "know" any more details than what B sends along.

Furtheremore, in DL this model of Theory B is scoped under a $\vdash_B$. So DL provides a (not unique but) simple set of constructs, concepts, and semantics of "containing" a possible inconsistency in Theory B, even while mixing knowledge from theory B with other propositions in a line of (machine executed) proof.

By Thomas Lord at Sat, 2015-06-27 16:49 | login or register to post comments

Runtime Checking

This only works in the case of mutual trust. If the code at one end can change, then the proof is useless as it is about a different version of the code. You need a runtime mechanism to do the type checking of messages sent to remote actors.

By Keean Schupke at Sat, 2015-06-27 17:49 | login or register to post comments

re runtime checking

> This only works in the case of mutual trust. If the code at one end can change, then the proof is useless as it is about a different version of the code.

More generally, the remote code can be wrong for any reason.

Hence slogans like: Distributed and decentralized information systems are pervasively inconsistent.

One question is what kind of system architectures can be robust given that they will (assumably) be pervasively inconsistent?

Another question is what kinds of mathematical and programming tools are useful for those kinds of architecture.

By Thomas Lord at Sat, 2015-06-27 18:06 | login or register to post comments

Decrypting and signature verifying are runtime checks

Decrypting and signature verifying are runtime checks.

By Hewitt at Sat, 2015-06-27 18:07 | login or register to post comments

Secure Binaries and Unsafe Remote Execution.

Yes, that's why they have a performance cost. Generally we can assume everything linked into a single binary is going to be compile time checked and can have security proofs.

What happens if I build a malicious library directly in assembly, I can fake the encryption signature to get at the data, but then treat it locally as different types?

Anything not in the same binary, like other application binaries, libraries etc need to be considered untrusted, just like services provided by another machine would be.

By Keean Schupke at Sat, 2015-06-27 19:05 | login or register to post comments

Process calculi security omission: aspects

There is a subtle security requirement that seems to be omitted from process calculi, namely, aspects of Actors.

An external party must not be able to automatically cast (as in Java) the address of an Actor to any internal interface that the Actor happens to implement.

By Hewitt at Tue, 2015-06-23 18:30 | login or register to post comments

"A calculus of communicating systems" [Milner 1986]

Robin Milner's 1980 book, "A calculus of communicating systems", presents a mature account of CCS which I gather is the formalism adopted by the topic paper. It was reprinted in 1986 by the University of Edinburgh and is conveniently available on-line (albeit, as a scan rather than a searchable text).

"A calculus of communicating systems" Milner, 1986

By Thomas Lord at Tue, 2015-06-23 20:17 | login or register to post comments

so far I like that paper (book)

The whole thing looks worth reading, and intro section **0.2 Character** is especially engaging, about the opposition of observing extensional behavior versus structure of intensional private definition. Many nice remarks summarize pithy notions, such as the only way to observe a system is to communicate with it.

Note opposition is an important idea in presenting ideas clearly, because to say what a thing is, you must also say what it is *not*. In particular, it's important to identify antithetical aspects that do *not* belong to members of a group of things you want to define. By default, mixin attributes are considered okay by most people, as long as they are not antagonistic aspects. Identifying antithetical ideas and explaining why they don't get along will improve clarity of whatever you want to define in terms of those ideas.

An amusing application of Murphy's Law to software is that whenever you define an object X as the antithesis of Y, you will come across requirements demanding you masquerade X as a Y, no matter how badly it fits. For example, you might find a way to do X very efficiently, as long as usage context Y never happens. Then folks will ask you to implement Y using X, even after you explain it has nearly pessimal performance despite giving the correct answer.

By Rys McCusker at Tue, 2015-06-23 21:36 | login or register to post comments

"A brief history of process algebra" [Baeten 2004]

J.C.M. Baeten wrote a kind of history-of-thought of process algebra.

"A brief history of process algebra" [Baeten 2004]

By Thomas Lord at Tue, 2015-06-23 20:20 | login or register to post comments

Milner on Actors vs. CCS.

In the introduction "A calculus of communicating systems" Milner surveys alternative approaches and says something interesting about the Actor approach:

> We now turn to two models based on non-synchronized communication. One, with strong expressive power, is Hewitt's Actor Systems; a recent reference is [MIT AI Memo 505]. Here the communication discipline is that each message sent by an actor will, after finite time, arrive at its destination actor; no structure over waiting messages (e.g. ordering by send-time) is imposed. This, together with the dynamic creation of actors, yields an interesting programming method. However, it seems to the author that the fluidity of structure in the model, and the need to handle the collection of waiting messages, pose difficulties for a tractable extensional theory.

It is interesting to note that that Hewitt-today and Milner-ca.-1980 are almost entirely in agreement except for the last part:

> However, it seems to the author that the fluidity of structure in the model, and the need to handle the collection of waiting messages, pose difficulties for a tractable extensional theory.

I would note here that Milners' CCS can be regarded as implying a kind of program structure discipline -- a style of program -- that can be straightforwardly realized using Actors and, in that realization of a CCS language, proved to satisfy Milners' axioms.

By Thomas Lord at Tue, 2015-06-23 20:31 | login or register to post comments

Please see my plea re: actor discussions

hi, Please **see my plea re: actor discussions**.

By raould at Tue, 2015-06-23 20:47 | login or register to post comments

re Please see my plea

> hi, Please see my plea re: actor discussions.

I have. I will respond to your plea in the separate obnoxious thread that you started.

Here, I am providing some background material to help you or someone else understand the book you say you don't think you understand.

Is that OK with you? To actually provide useful background and context about the book you turned into a topic? Do you mind?

By Thomas Lord at Tue, 2015-06-23 20:49 | login or register to post comments

unfortunately

the 'actor' angle on anything on LtU has gone past the breaking point for me. Probably having the first post on this thread be a rather non-sequiter from Hewitt pushing the discussion toward actors put me in a state of mind where any mention of actor by anybody here seems like just feeding trolls. Apologies if we are just very far apart at the moment due to our personal subjective experiences on LtU.

By raould at Tue, 2015-06-23 21:57 | login or register to post comments

Denotational semantics of Actor Model

Milner had a good point in his CCS memo. It took a long time to develop an adequate denotational semantics for the Actor Model (what Milner called a "tractable extensional theory"). Because of these difficulties, Milner imposed severe restrictions on CCS that made it inadequate as a general model of (concurrent) computation. To remedy some of these deficiencies in CCS, Milner later developed the pi-calculus (which was still not adequate for general concurrency).

The first adequate denotational semantics for the Actor Model was published here:
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.

The issue that must be addressed is how to denotationally model all of the messages that are "in flight."

By Hewitt at Tue, 2015-06-23 21:10 | login or register to post comments

"What is Commitment? Physical, Org'zational, and Social" [2006]

For convenience of readers:

"What is Commitment? Physical, Organizational, and Social" [Hewitt 2006]

By Thomas Lord at Tue, 2015-06-23 21:19 | login or register to post comments

aside re Denotational semantics of actor model

> The issue that must be addressed is how to denotationally model all of the messages that are "in flight."

Very funny.

By Thomas Lord at Tue, 2015-06-23 21:58 | login or register to post comments

"Spec. and Prov. Props. of Guardians for Dist. Sys" [Hewit 1979]

Ca. 1979, about the same time as "A calculus of communicating systems" [Milner 1986], Hewitt et al. published AI Memo 505. Milner cites this paper:

"Specifying and proving properties of guardians for distributed systems" [Hewitt, Attardi, Lieberman 1979]

The paper illustrates the actor paradigm through a worked, practical example with particular attention to the role of types, to the asynchronous model of communication, to the kind of proof methodology useful for what Milner called a "tractable extensional theory", and some hints about future automated reasoning about programs.

By Thomas Lord at Tue, 2015-06-23 20:41 | login or register to post comments

"Concurrency and automata on infinite sequences" [Park 1981]

The 1986 reprint of "A calculus of communicating systems" [Milner 1980] is basically unrevised from the earlier edition but for the addition of a new introdution.

The new introduction makes excited report of the work of D.M.R. Park for which the seminal introduction appears to be:

"Concurrency and automata on infinite sequences" [Park 1981]

I have not yet read much of this paper but it does leap out at me from the abstract and introduction that part of the point here is to explicate the theory that follows from a particular semantic interpretation of communicating non-deterministic automata that provides "unbounded nondeterminism".

Earlier attempts to generalize, for example, denotational semantics to concurrent systems ran into the problem that they did not model unbounded nondeterminism.

(The paper gives an example to provide an intuitive definition of what "unbounded nondeterminism" means. In the program:

```
x:= true; y := 1; ((while x do y := y+1) par x := false)
```

the semantics of the language might imply the program *could* never terminate, or might imply that the program always terminates but that the possible final value of y is unbounded. The latter case is called "unbounded nondeterminism".)

By Thomas Lord at Tue, 2015-06-23 21:10 | login or register to post comments

unbounded nondeterminism

It certainly comes up all the time. I've seen a group of very smart, talented, experienced people sit around a conference table talking about how weird 'servers' are because they never end so how can you prove anything about them, etc. So tools which could help arm people in that regard are appealing.

By raould at Tue, 2015-06-23 22:03 | login or register to post comments

CSP and ACP

Since this seems to be becoming a thread for posting links to introductory work on various process calculi, I'll throw in a few more here:

- Communicating Sequential Processes--Hoare's original book on CSP (differs somewhat from the 1978 paper of the same name). It is very readable, and presents a calculus that is somewhat like CCS but includes more syntactic sugar. Where CCS uses bisimilarity as an equivalence model, CSP typically uses less strict equivalences such as traces or failures refinement. CSP has, on the whole, had a more practical bent than CCS, and consequently seen more industrial use.
- The Theory and Practice of Concurrency--Roscoe (1997), a more modern treatment of CSP (last updated in 2005, now superceded by a newer book). A detailed look at both the semantic foundations of CSP and of its practical uses. Emphasises the use of the commercial FDR2 refinment checker (or model checker) for verifying that systems meet their specifications.
- "Foundational Calculi for Programming Languages"--Benjamin Pierce (1995) article that provides a beginner's introduction to the π-calculus. The title alone would make it seem appropriate for LtU.
- Introduction to Process Algebra--Fokkink (2007), an introduction to the Algebra of Communicating Processes (ACP), which, along with CCS and CSP, is one of the major process calculi from which many others have been derived.
- A gentle introduction to Process Algebras--Rocco De Nicola (2011) article that provides a great overview of process algebra (calculus) in general, the various kinds of process equivalences, an introduction to CCS, CSP, and ACP (the three most prominent process calculi), and and a discussion of how they relate to other models of concurrency

By Allan McInnes at Tue, 2015-06-23 22:46 | login or register to post comments

thanks to all

I guess I should have done a better job saying what I thought was particularly cool about the book. I am less hoping for intro stuff (although that is good, to spread the word, so ok thanks all!) and more about how of late the calculi have been used to prove things about e.g. unbounded nondeterminism. Things that might have seemed impossible for a long time. So that we can have our servers and eat them, too. So I would love to hear from those In The Know what the best state of the art is in process calculi. The particular tome I posted about seems to, according to the author with whom I've been in email contact, fallen into the dusty spaces. Are other approaches as successful in helping those who grok them to prove useful things about concurrency?

By raould at Tue, 2015-06-23 23:05 | login or register to post comments

re: Are other approaches as successful in helping those who grok

I'm sorry to that this might provoke you in the whole bogus anti-Hewittism but I will say that my enthusiasm for nearly everything I now realize he is saying is because I recognize that he has done most of the heavy lifting of formalizing how I and all the best hackers I know have believed about things for decades.

By Thomas Lord at Tue, 2015-06-23 23:19 | login or register to post comments

Please don't be discouraged

Thomas,

Please don't be discouraged.

Scientific discussions can become highly technical although it is important for practitioners to explain things as simply and intuitively as possible.

Insults and personal attacks are basically irrelevant in the long run although they can degrade LtU and you are right to fight against them.

It's important to concentrate on the argumentation because that's what is important for science and engineering discussion of the issues. **Science and engineering are not popularity contests.**

Also, as you have discerned, the Actor Model has become the default model of computation partly because it formalizes intuitions and practices that are becoming increasing relevant. Consequently, it provides terminology and principles for addressing many current issues.

Regards,
Carl

PS. Of course, I am not always right ;-)

By Hewitt at Tue, 2015-06-23 23:38 | login or register to post comments

I do think that is great

...and as I have previously written, I very much appreciate the work Hewitt has done as well (including his concerns and activism over the back-door-ness of our society, which is to me even more important than his actor work).

I am also glad that you are seeing this and getting into the depths of it. I'd appreciate you posting posts about what you've learned, in terms and in a writing style that others can engage with.

Unfortunately the style embraced by Hewitt has sort of been doing more harm than good here over time, I personally feel. That's, again, a sad thing since it is probably pushing people away from some good if not great work.

By raould at Tue, 2015-06-23 23:53 | login or register to post comments

Model checking (and TLA+)

Most of the industrial applications (if that's what you're interested in when you say "servers") of process calculus that I'm aware of have involved model checking rather than proof techniques.

Along the same lines, and unmentioned here so far, is Lamport's work on TLA+ (see here). TLA+ is neither a process calculus nor an Actor-based model. It is a temporal logic for describing the way in which the states of a system evolve. Lamport developed it specifically for specifying and proving things about concurrent systems. But, as with process calculi, although proof is possible it is model-checking that is getting more traction in industry. See, for example, the recent CACM article on Amazon's use of TLA+: How Amazon Web Services Uses Formal Methods by Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff
Communications of the ACM, Vol. 58 No. 4, Pages 66-73.

By Allan McInnes at Tue, 2015-06-23 23:29 | login or register to post comments

TLA+ low level: model checking explodes without abstraction

TLA+ is very low level. Without abstraction, model checking concurrent systems tends to explode with possibilities.

Is there anything more abstract than TLA+ that can be used for model checking?

By Hewitt at Tue, 2015-06-23 23:49 | login or register to post comments

State explosion

Yes, state explosion has long been a problem for model-checking. However, there are some interesting techniques (FDR2's compressions, SPIN's partial order reductions, and so on) that can help to combat it. Modern model-checkers can check systems orders of magnitude more complex than those checkable by earlier generations of model checkers.

That said, the best way to combat state explosion is, as you alluded, to choose an appropriate level of abstraction for the model. I agree that many of the examples of TLA+ that Lamport has shown are fairly low-level. But I don't think there's anything inherent in TLA+ that means that you can't operate at higher levels of abstraction. It is, after all, simply a logic over actions. The level of abstraction depends on what you define as "actions" in the system you're modeling. To a certain extent the same is true of other formalisms and model checkers, but seems especially true of TLA+ since it doesn't assume things like a process-based execution model or a specific kind of parallel composition (whereas, for example, model checkers for something like CSP are somewhat constrained by the fact that they are intended to check CSP-like systems).

I haven't looked at it in any detail, but you may be interested in this paper on modeling Actors in TLA+: "ReActor: A notation for the specification of actor systems and its semantics"

By Allan McInnes at Wed, 2015-06-24 01:34 | login or register to post comments

TLA+ incorrectly treats state as global

TLA+ incorrectly treats state as global, which is scientifically incorrect in addition to being disastrous for engineering.

PS. Thanks for the link to the ReActor article.

By Hewitt at Wed, 2015-06-24 12:54 | login or register to post comments

Global state

I'm not really all that familiar with TLA+, but my understanding is that it does allow the use of existential quantification to hide variables. I'm not sure whether or not that fully addresses your concerns though. You'll have to take that up with Lamport. Presumably he had his reasons for designing TLA+ the way he did.

By Allan McInnes at Wed, 2015-06-24 16:13 | login or register to post comments

This post was moved to start new topic

This post was moved to start a new topic.

By Hewitt at Tue, 2015-06-23 23:21 | login or register to post comments

great reading on concurrency: Lamport's seminal papers

Turns out the damn things are online. Here's another angle to understand how physics and concurrency meet up in maths. If you find special relativity kind of tickling and fun, you'll dig these papers.

BTW, the unrevised origins of these papers go back to 1980 so they are concurrent (heheh) with the matured appearance of CCS (Milner) and with Hewitt's 1979 on Guardians (linked elsewhere in the thread).

"The Mutual Exclusion Problem: Part I" [Lamport 2000]

"The Mutual Exclusion Problem: Part II" [Lamport 2000]

By Thomas Lord at Wed, 2015-06-24 00:21 | login or register to post comments

re pure and impure actors

> But even with lexically scoped actors, it does not seem trivial to determine if multiple threads can simultaneously call the actor.

In the *general case*, it is a turing incomplete problem.

(The general case doesn't have to be solved to write a good compiler.)

By Thomas Lord at Mon, 2015-06-29 17:56 | login or register to post comments

Solving some cases

If we can somehow represent threads/processes in the type system, we should be able to catch when an actor is potentially accessed from multiple threads, and insert a mutex.

The actor type system does not seem to be able to capture the purity of an actor, so it would need to be enhanced with monads or effects.

The actor type system does not seem to be able to capture the purity of an actor, so it would need to be enhanced with monads or effects.

If we enhanced the type system in those ways, I could see how a compiler could produce efficient code.

By Keean Schupke at Mon, 2015-06-29 20:10 | login or register to post comments

Cacheable Actors: special case that can be typed

Cacheable Actors are an important special case that can be typed.

For example, that Factorial is cacheable (i.e. caching is allowed) can be be expressed as follows:

```
Interface Factorial with [N] |••> N.
```

Monads are incompatible with general massive concurrency because they sequence computation steps. Types for borrowing can be awkward for massive concurrency because it can be difficult to keep track of which program is borrowing.

PS. It's probably preferable to mark the messages that cacheable than to mark the messages that are not cacheable although this is not a big deal. Note that a message may be determinate (same result from same request) and still not cacheable because the message can have effects.

PPS. This comment was edited while the following comment by Keean was being composed.

By Hewitt at Mon, 2015-06-29 20:31 | login or register to post comments

Type of a determinate actor

if we represent a pure function:

```
f :: a -> a
```

Then we can use a monad to indicate that the function is stateful:

```
f :: a -> IO a
```

If we consider these actors:

```
f.[a] -> a
f.[a] -> IO a
```

Now the monad means we have to serialise access, from a single thread it is no problem, and from multiple threads we need to insert a mutex.

So your statement "Monads types for borrowing are incompatible with general massive concurrency" is clearly a not a problem. Monads precisely indicate exactly those actors that we need mutex serialised access when used from multiple threads. So far from being incompatible with massive concurrency they provide exactly the type information you need in order to optimise the code appropriately.

Edit: They also provide the distinction between functions/actors that need initialisation (stateful) and those that do not. An alternative would be to consider all functions determinate, and handle stateful functions/actors like C++ function objects (of course the two are related, a stateful object being the fixed point of a record of monadic functions and the state).

By Keean Schupke at Mon, 2015-06-29 20:41 | login or register to post comments

Monads vs Marking

Monads allow the type system to automatically track statefullness. Marking functions as cacheable is not safe, as you can get it wrong. Much better to have the type system enforce it.

Also we are not interested in cacheability, but whether access had to be serialised, so the caching annotation does not do exactly what we want in any case.

I don't get the objections to monads with concurrency. If monads are used to contain statefullness they provide exactly the information needed to decide whether an actor needs a mutex for incoming messages (unlike cacheability which does not).

A non-monadic actor would be stateless and side-effectless (pure) and because local variables are stored on the stack (which is thread local) the actor can simply be jumped into (it is safe to use JSR and RTN). In this case passing a message to an actor would reduce to a plain function call.

In the monadic case the sender would need to obtain a lock on the receiver before jumping, but only if multiple threads try and send messages to the actor. One obvious pattern is a stateful actor and a message queue such that the stateful actor is only messaged from a single thread from the queue, but the queue would still need mutex protection. I am not sure yet how to track the number of threads that pass messages.

By Keean Schupke at Tue, 2015-06-30 05:22 | login or register to post comments

Cacheable message types must be checked where possible.

Cacheable message types must be checked where possible. If an Actor on another computer belonging to someone else has type with a message marked as cacheable, then response can be obtained from cache.

Monads (à la Scott and Strachey) in general mean exponentially slower performance for changing Actors, which is the price of basing all semantics on the lambda calculus.

By Hewitt at Tue, 2015-06-30 06:28 | login or register to post comments

Monads

I kind of see your point. Accessing a mutable store in the IO monad means all mutations have to have a single big mutex. This is not what we want. If however each stateful actor were in its own State monad we don't have this problem. We should get exactly what we want only one thread in each stateful actor at a time.

So it seems to me you might be confusing the IO monad with monads in general.

By Keean Schupke at Tue, 2015-06-30 07:34 | login or register to post comments

LC doesn't define a rewrite strategy

> Monads (à la Scott and Strachey) in general mean exponentially slower performance for changing Actors, which is the price of basing all semantics on the lambda calculus.

Ridiculous. Lambda calculus is a formalism which defines a rewrite system; i.e., it defines a syntax and rewrite rules. Mostly, after that, it is used to study type systems and various confluency properties.

LC doesn't define a rewrite strategy. You can *chose* a rewrite strategy for LC. Innermost-outermost, call-by-name/value, *massively concurrent* if you feel like.

The Actor Model seems to have somewhat of a syntax and no associated rewrite calculus. It doesn't have a formal operational model, therefore doesn't let itself be compared to other computational models. A meaningless comparison.

Reversely, it suffices to show that an LC with concurrent evaluation can model a process algebra faithfully. I wouldn't know why you would want to do that, but that at least seems possible.

The language Clean implements a pure functional language based on concurrent rewriting of an LC.

Please provide a syntax, a rewrite calculus, and a rewrite strategy for the Actor Model before making claims.

By marco at Wed, 2015-07-01 20:34 | login or register to post comments

Actor Model has axiomatic and denotational semantics

Te Actor Model has axiomatic and denotational semantics. ActorScript currently has published meta semantics.

By Hewitt at Wed, 2015-07-01 20:40 | login or register to post comments

Yah

Yah. That's what you claim, but I haven't found anything meaningful while scanning your papers. Meta-semantics isn't good enough.

By marco at Wed, 2015-07-01 20:41 | login or register to post comments

Meta semantics for ActorScript is on HAL

See pages 51-73 of ActorScript for meta semantics.

By Hewitt at Thu, 2015-07-02 02:34 | login or register to post comments

Approximate correctness of concurrent programs

I hope to read Mingsheng Ying's book someday to see how it creates a topology of *approximate correctness* of concurrent programs. This is something that I wouldn't ordinarily think is possible, at least in systems that can model instantaneous events and arbitrary computation.

For instance, as far as I can tell, the Actor model makes some unrealistic simplifying assumptions:

First, with Actors, a message is instantaneously sent and instantaneously received, which means that if we blink, we might miss an important part of the system's behavior. If a system is approximately correct, hopefully it will still look approximately correct if we blink.

The idea that two parts of the program are concurrent means that they exist side by side as each of them goes through its own updates. That is, they exist at the same time, and their change timelines overlap: Change over a simultaneous interval. If we never discuss any instantaneous events, we have simultaneity everywhere; fingernails grow simultaneously with each other, but none of them grows instantaneously.

Second, Actors seem to assume arbitrary computation, at least in the sense of unbounded nondeterminism.

The unbounded part in itself makes sense to me. So many times, we choose not to put a specific bound on termination because we want to compose black-box modules that don't know each other's specific timing costs. So many times, we choose not to put a bound on termination because we're only concerned about enforcing type soundness, not enforcing feasible computation.

What doesn't work is the idea that we'd *want* arbitrary computation in this context. Sure we'd like to check the correctness of a wide range of systems, but we're not going to get any notion of "approximate correctness" if we insist that the programs should be expressive enough to blow up from any small glitch in our approximation.

So if I were to try to develop a topology of approximate correctness of concurrent programs, first I would make sure not to give myself black boxes full of arbitrary computation. All the internal state that *would* be kept in those boxes would be open-faced, giving us a sort of global state for reasoning purposes. All the sophisticated computation would be unwound into a sequential and parallel composition of steps, with explicit state resources for any behaviors that need to be more dynamic than that. Ideally, neither the "steps" nor the seams between them would have instantaneous semantics. Sequential and parallel composition would bring me to something resembling a process calculus, and I would expect to get fruitful support from reading Mingsheng Ying's book.

(A lot of what I'm saying, especially about "if we blink," is related to David's goal of glitch freedom for continuous reactive RDP. David decries instantaneous events at "Why Not Events.")

By Ross Angle at Sun, 2015-07-05 21:33 | login or register to post comments

first page previous page