Mobile process calculi for programming the blockchain Documentation

Release 1.0

David Currin, Joseph Denman, Ed Eykholt, Lucius Gregory Mered

Jan 16, 2018

Contents:

David Currin, Joseph Denman, Ed Eykholt, Lucius Gregory Meredith

December 2016

CHAPTER 1

Introduction

Beginning back in the early ’90s with MCC’s ESS and other nascent efforts a new possibility for the development of distributed and decentralized applications was emerging. This was a virtual machine environment, running as a node in a connected network of such VM-based environments. This idea offered a rich fabric for the development of decentralized, but coordinated applications that was well suited to the Internet model of networked applications. The model of application development that would sit well on top of such a rich environment was a subject of intensive investigation.

Here we provide a brief, and somewhat idiosyncratic survey of several lines of investigation that converge to a single model for programming and deploying decentralized applications. Notably, the model adapts quite well to the blockchain setting, providing a natural notion of smart contract. As such, this summary can be taken as the design rationale for the RChain architecture, and the rholang model of blockchain-based smart contracts.

CHAPTER 2

Actors, Tuples and π

2.1 Rosette

At MCC the Carnot research group predicted the commercialization of the Internet a full decade before Netscape rose to fame. The Carnot group, however was focused on decentralized and distributed applications, and developed a network application node, called the Extensible Services Switch, or ESS, and the programming language Rosette for programming these nodes. In Rosette/ESS the model under investigation was the actor model. Here we are speaking quite specifically of an elaboration of the model developed by Carl Hewitt and refined by Gul Agha, who subsequently consulted on the design of Rosette.

The Rosette elaboration was striking in scope and elegance. Specifically, Rosette decomposes an actor into

- a mailbox (a queue where messages from the actor's clients arrive)
- a state (a tuple of values)
- a meta (a description of how to access values in the state in terms of more abstract keys)
- and a shared behavior object (a map from message types to code to run in response, roughly equivalent to a vtable in languages like C++)

An actor's processing consists of reading the next message in the mailbox, using the shared behavior object (or sbo) to determine which code to run in response to the message and then execute that code in an environment in which references to keys described in the meta are bound to locations in the state tuple. That code will principally send messages to other actors and possibly await for responses.

An actor provides a consistent view of state under concurrent execution via an update; that is, an actor does not process the next message in the mailbox until it has called update, and while the actor is processing the current message, the mailbox is considered locked and is queuing client requests. When an actor calls an update a new logical thread of activity is created to process the next message in the mailbox queue. That thread's view of the state of the actor is whatever is supplied to the update. The previous thread may still make changes to the state, but they are not observable to all subsequent threads processing subsequent messages in the mailbox, and hence to all subsequent client requests. This approach provides a scalable notion of concurrency that collapses to message arrival order non-determinism in single threaded containers, and expands when there is system level support to map the logical threads in some outer container (such as a VM, an operating system, or the hardware, itself).

Already, this is a much more deliberately articulated model of actors than exists in most industrial systems today (cf Scala's AKKA framework). What makes the Rosette model so much more elegant, however, is its total commitment to meta-level programming. In the Rosette model, everything is an actor. In particular, the mailbox of an actor is an actor, and that has, in turn, a mailbox, state, meta, and sbo. It is noteworthy that Rosette makes this fully reflective model with the possibility of infinite regress perform on par or better than languages like Java or C#.

In some sense, the structural reflection of actors in the Rosette model is echoed in languages like Java where classes are in turn objects that can be programmatically probed and enlisted in computation. However, Rosette takes the reflective principle a step further. The model offers not just structural reflection, but procedural reflection, by providing a notion of continuation reconciled with the concurrency inherent in the model. Specifically, just as a shift-block in a shift-reset style presentation of delimited continuations makes the awaiting continuation available to the code in the block, Rosette's reflective method makes the awaiting continuation available as a parameter to the method in the body of the method. More on this later.

To close out this brief summary note that Rosette was not merely structurally and procedurally reflective, but also lexically reflective. That is, all syntactic structure of programs in Rosette are also actors! The reflective infrastructure for this provides the basis for a hygienic macro system, support for embedded domain specific languages, and a host of other syntactic and symbolic processing features that many industrial languages still struggle to provide some 20 years after Rosette's inception.

2.2 Tuplespaces

Around the same time as the Rosette model was being investigated for developing applications in this decentralized and distributed setting, Gelernter proposed Tuplespaces. Here the idea is to create a logical repository above the physical and communications layers of the Internet. The repository is essentially organized as a distributed key-value database in which the key-value pairs are used as communication and coordination mechanisms between computations. Gelernter describes three basic operations for interaction with the tuplespace, `out` to create a new data object in tuplespace, `in` to consume (remove) an object from tuple space and `rd` to read an object without removing it

In contrast to message passing systems this approach allows senders and receivers to operate without any knowledge of each other. When a process generates a new result that other processes will need, it simply dumps the new data into tuplespace. Processes that need data can look for it in tuple space using pattern matching.

```
out("job", 999, "abc")

in("job", 1000, ?x) ; blocks because there is no matching tuple
in("job", 999, x:string) ; takes that tuple out of tuple space, assigning 'abc' to x
```

This raises questions about how publication of data is persisted and what happens to computations suspended waiting on key-value pairs that are not present in the tuplespace. Moreover, the tuplespace mechanism makes no commitment to any programming model. Agents using the tuplespace may be written in a wide variety of programming models with a wide variety of implementations and interpretations of the agreed concurrency semantics. This makes reasoning about the end-to-end semantics of an application made of many agents interacting through the tuplespace much, much harder. However, the simplicity of the communication and coordination model has enjoyed wide appeal and there were many implementations of the tuplespace idea over the ensuing decades.

A notable difference between the tuplespace notion of coordination and the actor model lies in the principal port limitation of actors. An actor has one place, its mailbox, where it is listening to the rest of the world. Real systems and real system development require that applications often listen to two or more data sources and coordinate across them. Fork-join decision procedures, for example, where requests for information are spawned to multiple data sources and then the subsequent computation represents a join of the resulting information streams from the autonomous data sources, are quite standard in human decision making processes, from loan processing, to the review of academic papers. Of course it is possible to arrange to have an actor that coordinates amongst multiple actors who are then charged with handling the independent data sources. This introduces application overhead and breaks encapsulation as the actors need to be aware they are coordinating.

In contrast, the tuplespace model is well suited to computations that coordinate across multiple autonomous data sources.

2.3 Distributed implementations of mobile process calculi

Tomlinson, Lavender, and Meredith, among others, provided a realization of the tuplespace model inside Rosette/ESS as a means to investigate the two models side-by-side and compare applications written in both styles. It was during this work that Meredith began an intensive investigation of the mobile process calculi as yet a third alternative to the actor model and the tuplespace model. One of the primary desiderata was to bridge between having a uniform programming model, such as the actor model of Rosette, making reasoning about application semantics much easier, with the simple, yet flexible notion of communication and coordination afforded in the tuplespace model.

In the code depicted below the method names consume and produce are used instead of the traditional Linda verbs `in` and `out`. The reason is that once reflective method strategy was discovered, and then refined using delimited continuations, this lead to new vital observations relating to the life cycle of the data and continuation.

Listing 2.1: A Rosette implementation of the tuplespace get semantics

```
(defRMethod NameSpace (consume ctxt & location)
 ;;; by makng this a reflective method - RMethod - we gain access to the awaiting␣
↪continuation
 ;;; bound to the formal parameter ctxt
 (letrec [[[channel ptrn] location]
                ;;; the channel and the pattern of incoming messages to look for are␣
↪destructured and bound
        [subspace (tbl-get chart channel)]
                ;;; the incoming messages associated with the channel are collected␣
↪in a subtable
                ;;; in this sense we can see that the semantic framework supports a␣
↪compositional
                ;;; topic/subtopic/subsubtopic/... structuring technique that unifies␣
↪message passing
                ;;; with content delivery primitives
                ;;; the channel name becomes the topic, and the pattern structure␣
↪becomes
                ;;; the subtopic tree
                ;;; this also unifies with the URL view of resource access
       [candidates (names subspace)]
       [[extractions remainder]
          (fold candidates
            (proc [e acc k]
                (let [[[hits misses] acc]
                [binding (match? ptrn e)]]
            (if (miss? binding)
                (k [hits [e & misses]])
                (k [[[e binding] & hits] misses]))))))]
                  ;;; note that this is generic in the match? and miss? predicates
                  ;;; matching could be unification (as it is in SpecialK) or it␣
↪could be
                  ;;; a number of other special purpose protocols
                  ;;; the price for this genericity is performance
                  ;;; there is decent research showing that there are hashing␣
↪disciplines
                  ;;; that could provide a better than reasonable approximation of␣
↪unification
       [[productions consummation]
```

```
            (fold extractions
              (proc [[e binding] acc k]
                (let [[[productions consumers] acc]
               [hit (tbl-get subspace e)]]
                  (if (production? hit)
               (k [[[[e binding] hit] & productions] consumers])
               (k [productions [[e hit] & consumers]]))))))]]
                  ;;; this divides the hits into those matches that are data and
                  ;;; those matches that are continuations
                  ;;; and the rest of the code sends data to the awaiting continuation
                  ;;; and appends the continuation to those matches that are currently
                  ;;; data starved
                  ;;; this is a much more fine-grained view of excluded middle

   (seq
     (map productions
       (proc [[[ptrn binding] product]]
            (delete subspace ptrn)))
     (map consummation
          (proc [[ptrn consumers]]
            (tbl-add subspace
            ptrn (reverse [ctxt & (reverse consumers)]))))
     (update!)
     (ctxt-rtn ctxt productions))))
```

Listing 2.2: A Rosette implementation of the tuplespace put semantics

```
;;; This code is perfectly dual to the consumer code and so all the comments
;;; there apply in the corresponging code sites
(defRMethod NameSpace (produce ctxt & production)
 (letrec [[[channel ptrn product] production]
        [subspace (tbl-get chart channel)]
       [candidates (names subspace)]
       [[extractions remainder]
          (fold candidates
            (proc [e acc k]
                (let [[[hits misses] acc]
                [binding (match? ptrn e)]]
            (if (miss? binding)
                (k [[e & hits] misses])
                (k [hits [e & misses]]))))))]
       [[productions consummation]
            (fold extractions
              (proc [[e binding] acc k]
                (let [[[productions consumers] acc]
               [hit (tbl-get subspace e)]]
                  (if (production? hit)
               (k [[[e hit] & productions] consumers])
               (k [productions [[[e binding] hit] & consumers]]))))))]]
   (seq
     (map productions
       (proc [[ptrn prod]] (tbl-add subspace ptrn product)))
     (map consummation
       (proc [[[ptrn binding] consumers]]
       (seq
            (delete subspace ptrn)
            (map consumers
              (proc [consumer]
                (send ctxt-rtn consumer [product binding])
                binding)))))
     (update!)
     (ctxt-rtn ctxt product))))
```

Essentially, the question is what happens to either or both of data and continuation after an input request meets an output request. In traditional tuplespace and π-calculus semantics both data and continuation are removed from the store. However, it is perfectly possible to leave either or both of them in the store after the event. Each independent choice leads to a different major programming paradigm.

Traditional DB operations

Removing the continuation but leaving the data constitutes a standard database read:

	ephemeral - data	persistent - data	ephemeral - data	persistent - data
	ephemeral - k	ephemeral - k	persistent - k	ephemeral - k
producer	put	**store**	publish	publish with history
consumer	get	**read**	subscribe	subscribe

Traditional messaging operations

Removing the data, but leaving the continuation constitutes a subscription in a pub/sub model:

	ephemeral - data	persistent - data	ephemeral - data	persistent - data
	ephemeral - k	ephemeral - k	persistent - k	ephemeral - k
producer	put	store	**publish**	**publish with history**
consumer	get	read	**subscribe**	**subscribe**

Item-level locking in a distributed setting

Removing both data and continuation is the standard mobile process calculi and tuplespace semantics:

	ephemeral - data	persistent - data	ephemeral - data	persistent - data
	ephemeral - k	ephemeral - k	persistent - k	ephemeral - k
producer	**put**	store	publish	publish with history
consumer	**get**	read	subscribe	subscribe

Building on Tomlinson's insights about the use of Rosette's reflective methods to model the tuplespace semantics (see code above), Meredith provided a direct encoding of the π-calculus into tuplespace semantics via linear continuations. This semantics was at the heart of Microsoft's BizTalk Process Orchestration Engine, and Microsoft's XLang, arguably the first Internet scale smart contracting language, was the resulting programming model. This model was a direct influence on W3C standards, such as BEPL and WS-Choreography, and spawned a whole generation of business process automation applications and frameworks.

As with the refinements Rosette brings to the actor model, the π-calculus brings a specific ontology for applications built on the notion of processes that communicate via message passing over channels. It is important to note that the notion of process is parametric in a notion of channel, and Meredith used this level of abstraction to provide a wide variety of channel types in XLang, including bindings to Microsoft's MSMQ message queues, COM objects, and many other access points in popular technologies of the time. Perhaps most central to today's Internet abstractions is that URIs provide a natural notion of channel that allows for a realization of the programming model over URI aware communications protocols, such as http. Likewise, in terms of today's storage climate, keys in a key-value store, such as a nosql database also map directly to the notion of channel in the π-calculus, and Meredith used this very idea to provide the encoding of the π-calculus into tuplespace semantics.

2.3.1 From Tuplespaces to π-calculus

The π-calculus captures a core model of concurrent computation built from message-passing based interaction. It plays the same role in concurrent and distributed computation as the lambda calculus plays for functional languages and functional programming, setting out the basic ontology of computation and rendering it to a syntax and semantics in which calculations can be carried out. Given some notion of channel, it builds a handful of basic forms of process, the first three of which are about I/O, describing the actions of message passing.

- `0` is the form of the inert or stopped process that is the ground of the model
- `x?( ptrn )P` is the form of an input-guarded process waiting for a message on channel `x` that matches a pattern, ptrn, and on receiving such a message will continue by executing `P` in an environment where any variables in the pattern are bound to the values in the message
- `x!( m )` is the form of sending a message, `m`, on a channel `x`

The second three are about the concurrent nature of processes, the creation of channels, and recursion.

- `P|Q` is the form of a process that is the parallel composition of two processes P and Q where both processes are executing concurrently
- `(new x)P` is the form of a process that executes a subprocess, P, in a context in which x is bound to a fresh channel, distinct from all other channels in use
- `(def X( ptrn ) = P)[ m ]` and `X( m )`, these are the process forms for recursive definition and invocation

These basic forms can be interpreted in terms of the operations on Tuplespaces:

```
P,Q ::=                     [[-]](-) : π -> Scala =
    0                       { }
    | x?(prtn)P             { val ptrn = T.get([[x]](T)); [[T]](P) }
    | x!(m)                 T.put([[x]], m)
    | P|Q                   spawn{ [[P]](T)  }; spawn{ [[P]](T) }
    | (new x)P              { val x = T.fresh("x"); [[P]](T) }
    | (def X(ptrn) = P)(m)  object X { def apply(ptrn) = { [[P]](T) } }; X(m)
    | X(ptrn)               X(ptrn)
```

2.3.2 Monadically structured channel abstraction

Meredith then pursued two distinct lines of improvement to these features. Both of them are related to channel abstraction. The first of these relates the channel abstraction to the stream abstraction that has become so popular in the reactive programming paradigm. Specifically, it is easy to prove that a channel in the asynchronous π-calculus corresponds to an unbounded and persistent queue. This queue can be viewed as a stream, and access to the stream treated monadically, as is done in the reactive programming paradigm. This has the added advantage of providing a natural syntax and semantics for the fork-join pattern so prevalent in concurrent applications supporting human decision making applications mentioned previously.

```
( let [[data (consume ns channel pattern)]] P)
```

```
for( data <- ns.consume(channel, pattern) ){ P }
```

This point is worth discussing in more detail. While the π-calculus does resolve the principle port limitation of the actor model, it does not provide natural syntactic or semantics support for the fork-join pattern. Some variants of the π-calculus, such as the join calculus, have been proposed to resolve this tension, but arguably those proposals suffer an entanglement of features that make them unsuited to many distributed and decentralized programming design patterns. Meanwhile, the monadic interpretation of the channel provides a much more focused and elementary refactoring of the π-calculus semantics, consistent with all existing denotational semantics of the model, that provides a natural notion of fork-join while also mapping cleanly onto the reactive programming paradigm, and thus making integration of development stacks, such as Apache Spark, relatively simple.

If we look at this from the perspective of programming language evolution, we first see a refactoring of the semantics to look like:

```
P,Q ::=                      [[-]](-) : π -> Scala =
    0                        { }
    | x?(prtn)P              for( ptrn <- [[x]](T) ){ [[P]](T) }
    | x!(m)                  T.put([[x]], m)
    | P|Q                    spawn{ [[P]](T)  }; spawn{ [[P]](T) }
    | (new x)P               { val x = T.fresh("x"); [[P]](T) }
    | (def X(ptrn) = P)(m)   object X { def apply(ptrn) = { [[P]](T) } }; X(m)
    | X(ptrn)                X(ptrn)
```

where the for-comprehension is syntactic sugar for a use of the continuation monad. The success of this interpretation suggests a refactoring of the **source** of the interpretation.

```
P,Q :: = 0
         | for (ptrn <- x)P
         | x!(m)
         | P|Q
         | (new x)P
         | (def X(ptrn) = P)[m]
         | X(ptrn)
```

This refactoring shows up in Meredith and Stay's work on higher categorical semantics for the π-calculus *[SM15]*, and is then later incorporated in the rholang design. The important point to note is that the for-comprehension-based input can now be smoothly extended to input from multiple sources, each/all of which must pass a filter, before the continuation is invoked.

$$for(ptrn_1 \leftarrow x_1; \ldots ; ptrn_n \leftarrow x_n if cond)P$$

Using a for-comprehension allows the input guard semantics to be parametric in the monad used for channels, and hence the particular join semantics can be supplied polymorphically. The significance of this cannot be overemphasized. Specifically:

- It contrasts with the join-calculus where the join is inseparably bound together with recursion. The monadic input guard allows for anonymous, one time joins, which are quite standard in fork-join patterns in human decision processes.
- It provides the proper setting in which to interpret Kiselyov's LogicT monad transformer. Searching down each input source until a tuple of inputs that satisfies the conditions is found is sensitive to divergence in each input source. Fair interleaving, and more importantly, a means to programmatically describe interleaving policy is

critical for reliable, available, and performant services. This is the actual import of LogicT and the right setting in which to deploy that machinery.

- We now have a syntactic form for nested transactions. Specifically, `P` can only run in a context in which all of the state changes associated with the input sources and the condition are met. Further, `P` can be yet another input-guarded process. Thus a programmer, or a program analyzer, can detect transaction boundaries *syntactically*. This is vital for contracts involving financial and other mission-critical transactions.

2.3.3 A pre-RChain model for smart contracts

This is a precursor to the RChain model for smart contracts, as codified in the rholang design. It provides the richest set of communication primitives for building contracts proposed to date that has been driven both by theory and by industrial scale implementation and deployment. Yet, the entire set of contract primitives fits on a single line. There is not a single design proposal in this space, from the PoW-based blockchain to the EVM, that meets the quality assurance pressures this proposal has withstood. Specifically, the proposal folds in all the experiences using Rosette, Tuplespaces, and BizTalk and boils them down to a single design that meets the desiderata discovered in all of these efforts. It does so with only seven primitives, and primitives that line up with the dominant programming paradigms of the current market. Yet, as the examples from the rholang spec, and the paper on preventing the DAO bug with behavioral types show, the entire range of contracts expressible in existing blockchain technology is compactly expressed in this model.

As seen in the rholang design, however, this is only the beginning of the story. A little background is necessary to understand the import or this development. For the last 20 years a quiet revolution has been going on in computer science and logic. For many years it was known that for small, but growing fragment of the functional programming model types corresponded to propositions, and proofs corresponded to programs. If the correspondence, known variously as the proposition-as-types paradigm or the Curry-Howard isomorphism, could be made to cover a significant, practical portion of the model, it has profound implications for software development. At a minimum it means that the standard practice of type-checking programs coincides with proofs that programs enjoy certain properties as a part of their execution. The properties associated with the initial fragment covered by the Curry-Howard isomorphism largely had to do with respecting the shape of data flowing into and out of functions, effectively eliminating certain class of memory access violations by compile time checks.

With the advent of J-Y Girard's linear logic, we have seen a dramatic expansion of the proposition-as-types paradigm. With linear logic we see the expansion of the coverage far beyond the functional model, which is strictly sequential. Instead, the coverage offered by type checking for proving properties extends to protocol conformance checks in concurrent execution. Then Caires and Cardelli discovered the spatial logics which further expanded the coverage to include structural properties of the programs internal shape. Building on these discoveries, Stay and Meredith identified an algorithm, the LADL algorithm, for generating type systems such that well typed programs would enjoy a wide variety of structural and behavioral properties ranging from safety and liveness to security properties. By the application of the LADL algorithm developed by Stay and Meredith, this untyped model of the contract primitives identified here can be given a sound and complete type system rich enough to provide compile time safeguards that ensure key safety and liveness properties expected of mission-critical applications for handling financial assets and other sensitive content. A single example of such a compile time safeguard is sufficient to have caught and prevented the bug that led to the loss of 50M USD from the DAO, at compile time.

SpecialK

The monadic treatment of channel semantics is the insight explored in the SpecialK stack. Firstly, it maps channel access to for-comprehension style monadically structured reactive programming. Secondly, it maps channels simultaneously to local storage associated with the entire node, as well as to queues in an AMQP provider based communication infrastructure between nodes. This provides the basis of a content delivery network that can be realized over a network of communicating nodes, that is integrated with a π-calculus based programming model. In particular, as can be seen in the comments in the code above, the monadic treatment of channel + pattern unifies message-passing and content delivery programming paradigms. Specifically, the channel can be seen as providing topic, while the pattern provides nested subtopic structure to the message stream. This integrates all of the standard content addressing mechanisms, such as URLs + http, as well as providing a query model. See the section below for details.

From SpecialK to RChain

As we will see, the RChain model for contracts inherits all of SpecialK's treatment of content delivery. Yet, where SpecialK realized the pre-RChain contract model as an embedded domain specific language hosted as a set of libraries in Scala, the RChain model realizes the model as a full blown programming language to be run on a VM replicated on the blockchain, very much in the spirit of Ethereum's architecture and design. This choice addresses several shortcomings in the Synereo V1 architecture as outlined in the first Synereo white paper. In particular, it avoids the problem of having to pay other blockchains fees to run the financial capabilities of the attention economy, and thus suffering a number of economics-based attacks on the attention economy system contracts. It also addresses technical debt in the SpecialK stack related to the Scala delimited continuations library central to the SpecialK semantics, while dramatically increasing the capability of the smart contracts supported.

2.3.4 Rho-calculus

While the monadic abstraction provides structure on the stream of content flowing over channels a more fundamental observation provides the necessary structure to support industrial scale meta-level programming. It is important to recognize that virtually all of the major programming languages support meta-level programming. The reason is simply fact that programmers don't write programs. Programs write programs. Programmers write the programs that write programs. This is how the enormous task of programming at Internet scale is actually accomplished, using computers to automate as much of the task as possible. From text editors to compilers to code generators to AI, this is all a part of the basic ecosystem that surrounds the production of code for services that operate at Internet scale.

Taking a more narrow perspective, it is useful to witness the painful experiences of Scala to add support for meta-level programming after the fact of the language design. Reflection in Scala was not even thread safe for years. Arguably, this experience, plus the problems with the type system were the reasons for the back-to-the-drawing board effort underlying the dotty compiler and new language design. These and other well explored efforts make it clear that providing primitives for meta-level programming from the outset of the core design of the programming model is essential for longevity and practical use. In short, a design that practically supports meta-level programming is simply more cost effective in a project that wants to get to production-ready feature set on par with say Java, C#, or Scala.

Taking a cue from Rosette's total commitment to meta-level programming, the **r**-eflective **h**-igher **o**-rder π-calculus, or rho-calculus, for short, introduces reflection as part of the core model. It provides two basic primitives, reflect and reify, that allow an ongoing computation to turn a process into a channel, and a channel that is a reified process back into the process it reifies. The model has been peer reviewed multiple times over the last ten years. Prototypes providing a clear demonstration of its soundness have been available for nearly a decade. This takes the set of contract building primitives to a grand total of nine primitives, far fewer than found in Solidity, Ethereum's smart contracting language, yet the model is far more expressive than Solidity. In particular, Solidity-based smart contracts do not enjoy internal concurrency.

2.4 Implications for resource addressing, content delivery, query, and sharding

Before diving into how the model relates to resource addressing, content delivery, query and sharding, let's make a few quick observations about path-based addressing. Note that paths don't always compose. For example, take */a/b/c* and */a/b/d*. These don't compose naturally to yield a path. However, every path is automatically a tree, and as trees these do compose to yield a new tree */a/b/c+d*. In other words, trees afford a composable model for resource addressing. This also works as a query model. To see this latter half of this claim let's rewrite our trees in this form:

$$/a/b/c \mapsto a(b(c))$$

$$/a/b/c + d \mapsto a(b(c,d))$$

Then notice that unification works as a natural algorithm for matching and decomposing trees, and unification-based matching and decomposition provides the basis of query.

In light of this discussion, let's look at the I/O actions of the π-calculus:

```
input: x?(a(b(X,Y)))P  for(a(b(X,Y)) <- x)P
output: x!(a(b(c,d)))
```

When these are placed in concurrent execution we have:

```
for(a(b(X,Y)) <- x)P | x!(a(b(c,d)))
```

which evaluates to `P{ X := c, Y := d }`, that is we begin to execute `P` in an environment in which `X` is bound to `c`, and `Y` is bound to `d`. We write the evaluation step symbolically:

```
for(a(b(X,Y)) <- x)P | x!(a(b(c,d))) → P{ X := c, Y := d }
```

This gives rise to a very natural interpretation:

- Output places resources at locations:

```
x!(a(b(c,d)))
```

- Input queries for resources at locations:

```
for(a(b(X,Y)) <- x)P
```

This is only the beginning of the story. With reflection we admit structure on channel names, like x in the example above, themselves. This allows to subdivide the space where resources are stored via namespaces. Namespaces become the basis for a wide range of features from security to sharding.

2.4.1 The RChain model of smart contracts

Now we have a complete characterization of the RChain model of smart contracts. It is codified in the rholang design. The number of features it enjoys as a result of reflection alone, from macros to protocol adapters, is enough to warrant consideration.Taking a step back, however, we see further that

- it enjoys a sound and correct type system
- a formal specification
- a rendering of the formal specification to working code
- it dictates a formal specification of a correct-by-construction VM
- this dictates a clear compilation strategy as a series of correct-by-construction transforms to the byte code for a VM that has been field test for 20 years

Now compare this starting point to Ethereum's current point with Solidity and the EVM. If the goal is to produce a believable timeline over which we reach a network of blockchain nodes running formally verified, correct-by-construction code, then even with Ethereum's network effect this approach has distinct advantages. Clearly, there is enough market interest to support the development of both options.

CHAPTER 3

Enter the blockchain

One simple way to interpret blockchain technology is that it provides a decentralized replication technology. State can be replicated across a network of nodes without a centralized authority managing the replication. The subtle interplay between this replication mechanism and economics-based security sometimes obscures the basic value of the replication, itself. Replication of this nature is already a high impact value proposition. Witness Ethereum's proposal. The essence of it is to replicate the state of a virtual machine, instead of the state of ledger, on the blockchain. This provides not just a smart contracting system, but a global public compute resource that cannot be taken down.

More than Wall St or the City should be sitting up and taking notice. Amazon's AWS, Google's Cloud Service, Microsoft's Azure, all of these businesses become profoundly affected by the Ethereum proposal. The fact that the ledger, or economic aspect of replicated state can now be used as an economic security measure against DoS attacks on the global compute resource serves to strengthen the proposal. It also changes the nature of the ecosystem that uses these services.

The entire value proposition of cloud service infrastructure providers vitally depends on microtransactions linked to the use of compute resources. The Ethereum proposal integrates this at lowest level of execution and storage, in a manner that is crucial to the security of the system. It's simply a better, more elegant design than incumbent cloud solutions. More importantly, it's realized as a public resource. One of the recent revolutions that shares some characteristics of this emerging situation is the shift in telecommunications as a result of the Internet. The telephony providers knew that the Internet was going to change the way telecommunications was done and had to scramble to adapt and adjust to a landscape where more than half of long distance telecommunications would be conducted using free services like Skype, zoom, and Google Hangouts. Cloud infrastructure service providers, and the whole interconnected network of services running on them will likewise have to shift to a landscape where there is a sophisticated cryptocurrency market integrated with globally available, public compute resources at the lowest level.

3.1 Casper

At the 20K foot level this is a profound idea, but at the level of execution for production systems, it highlights several desiderata. First of all, it requires a different replication technology! The very technology that gave birth to the Ethereum proposal is the first thing that has to be re-imagined to make Ethereum's idea viable at Internet scale for production systems. Proof of work is simply too costly in terms of energy. The arguments for this are numerous, and all of the major stakeholders in Ethereum's development community have embraced them. Instead, a new pay-to-play algorithm, called Casper, is under development by a loose federation of stakeholders, including Buterin, Zamfir, and Meredith.

3.1.1 Bet by block versus bet by proposition

At the core of the Casper consensus algorithm is a cycle of message exchange amongst validators participating in the replication protocol. This cycle of message exchange ultimately converges to produce the consensus state. The protocol is factored in such a way that the consensus cycle, sometimes called the betting cycle, works independently of the type of state on which the validators are developing a consensus. It could be used to develop consensus on a single boolean value (e.g. the result of a coin flip), or something much more sophisticated, like the state of a virtual machine. This is further articulated into how the state is packaged. Specifically, the very name, blockchain, refers to the fact that state is packaged in blocks that are chained together. Thus, a very natural object of the consensus cycle is a block. Roughly speaking, validators exchange messages to converge on the next block in the chain.

This represents a natural, step-by-step evolution from PoW algorithms. Notice, however, that this rate limits block production to the rate of consensus cycle. Because blocks constitute a packaging of transactions (txn's for short), this provides a rough estimate of the transaction processing capability of the algorithm. Vitalik's PoC suggests that this is much, much faster than PoW-based BTC blockchain performance. However, the aim is to get into the range of 40K txns/sec, roughly on par with the transaction rates of financial networks like Visa. To achieve this we make two fundamental observations. The first of these is common sense: most transactions are isolated. The transaction associated with the person buying car parts in Shanghai is separate and isolated from the transaction associated with the person buying coffee in Seattle. Isolation of multiple concurrent transactions follows the same principle as the multi-lane freeway. As long as there's little to no lane-changing, there's a much greater throughput on a multi-lane freeway than on a single lane highway. No lane-changing is what we mean by isolation. Following this analogy, we want a block to represent a section of highway with multiple lanes for which there is virtually no lane-changing.

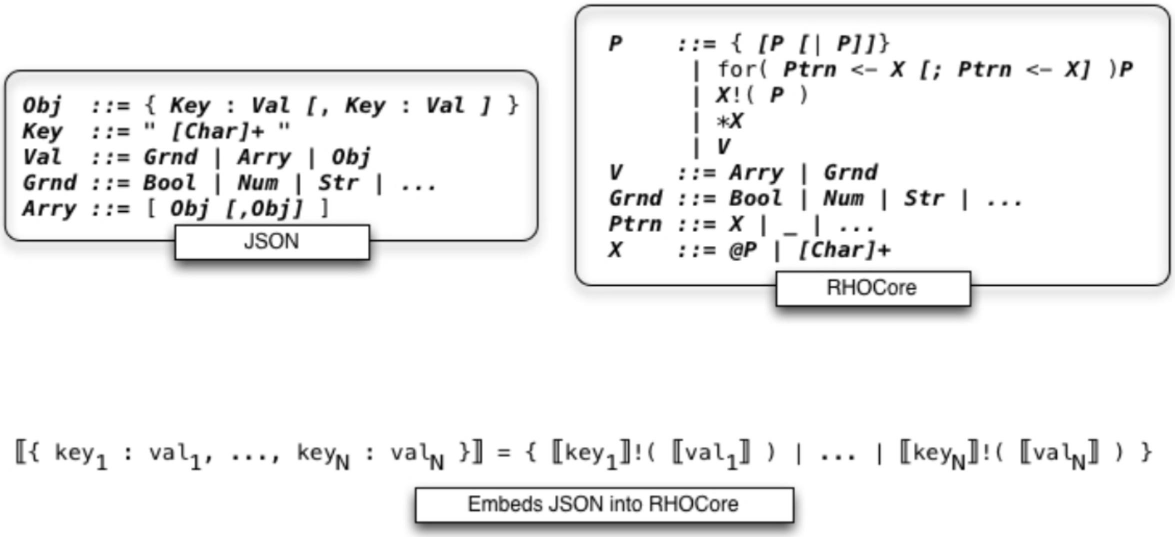
The other observation is more subtle. It has to do with programmatic compression. One analogy that might help comes from computer graphics. For many images, such as that of a fern, there are two fundamentally different ways to draw the image. In one of them, a large set of data points corresponding to pixels is provided to the renderer. Presumably, this data set is provided by sampling, such as might be provided by scanning an actual fern. The other way is to recognize that there is an algorithmic pattern associated with ferns that is essentially captured by a fractal. Then the data set of pixels can be represented in a tiny program that generates the fractal image corresponding to the fern. In this sense the program represents a form of compression. Rather than ship the large data set of pixels to the renderer, a relatively small program is shipped instead, and the renderer runs the program to get a stream of pixels it can pull as fast as it can draw.

Collections of transactions can be given similar compression using propositions. The LADL algorithm of Stay and Meredith describes precisely how a proposition corresponds to a collection of transactions. What remains is how to relate different propositions proposed by different validators. Specifically, suppose validator Abed is looking at transactions arising from one portion of the network while validator Troy is looking at transactions arising from another portion of the network. Assuming Abed and Troy are communicating about those groups of transactions using propositions, how do we make sure that the groups of transactions are consistent? This is where propositions as interpreted by the LADL algorithm have significant compression power.

By reasoning via the logical level it is possible to determine whether a collection of propositions is consistent. In the case of propositions interpreted by the LADL algorithm they will be consistent if and only if the meaning of their combination (say conjunction) (i.e. the collection they denote) is not empty. When the combination is empty then one or more of the propositions is inconsistent. At this point the validators invoke the Casper betting cycle to pick a winner amongst the maximally consistent subsets of propositions. Once the betting cycle converges, the block that is determined is given in terms of a collection of consistent propositions which then expands out to a much, much larger set of transactions, much the same way that the fern is rendered from the fractal decompression algorithm that produces the pixel set for the image.

3.1.2 From data storage to block storage in the RChain model

Building on the observations in the section on content storage and query we can detail a direct representation of JSON into a fragment of the rholang syntax referred to in the diagram below as RHOCore.



```
Obj  ::= { Key : Val [, Key : Val ] }
Key  ::= " [Char]+ "
Val  ::= Grnd | Arry | Obj
Grnd ::= Bool | Num | Str | ...
Arry ::= [ Obj [,Obj] ]
```

```
P    ::= { [P [| P]]}
       | for( Ptrn <- X [; Ptrn <- X] )P
       | X!( P )
       | *X
       | V
V    ::= Arry | Grnd
Grnd ::= Bool | Num | Str | ...
Ptrn ::= X | _ | ...
X    ::= @P | [Char]+
```

$$[\![\{\ \mathtt{key}_1 : \mathtt{val}_1, \ldots, \mathtt{key}_N : \mathtt{val}_N\ \}]\!] = \{\ [\![\mathtt{key}_1]\!]!(\ [\![\mathtt{val}_1]\!]\ ) \mid \ldots \mid [\![\mathtt{key}_N]\!]!(\ [\![\mathtt{val}_N]\!]\ )\ \}$$

Fig. 3.1: RSON : RChain data representation

This illustrates a meta-level point about the data storage interpretation. The storage and access semantics is consistent with a self-hosted serialization format, for rendering state to disk or wire. in practical terms, whatever a programmer has rendered to JSON for serialization on the wire or storage in a database like mongodb, can be rendered to an isomorphic expression in a fragment of the rholang syntax; and that expression, if it were executed, would effect the storage. Moreover, the complexity of the format exactly mirrors JSON. However, the spatial types of rholang serve to provide a type directed data validation mechanism for serializing and deserializing the data.

However, this only uses the output or producer side of the representation. By including the input or consumer side of the representation we can also provide a faithful and efficient representation of block structure. First note the meaning of the input guarded form. The continuation is guaranteed to execute in a context in which the values have been observed at the channels.

This is precisely a transactional guarantee. From this we can create a faithful interpretation of block structure that corresponds precisely to program syntax.

3.2 Sharding

Another desideratum in making the Ethereum proposal practical is to ditch the global computer! Instead of a single VM running on the blockchain, what is required is a composition of VMs each serving a shard of client processing. In some sense this marks a return to the original vision of the Internet as conceived when the Rosette/ESS design was proposed. There are some key differences, however. First, the state of each VM is stored on the blockchain. Second, though each VM is cut of the same cloth there is a discipline governing how they interact. Specifically, though they are all effectively copies of the same VM, each is operating on specific virtual address spaces, or namespaces as we have been calling them. When they are operating on the same namespace we have the guarantee that the state across each copy is exactly the same. This is what the consensus algorithm is for.

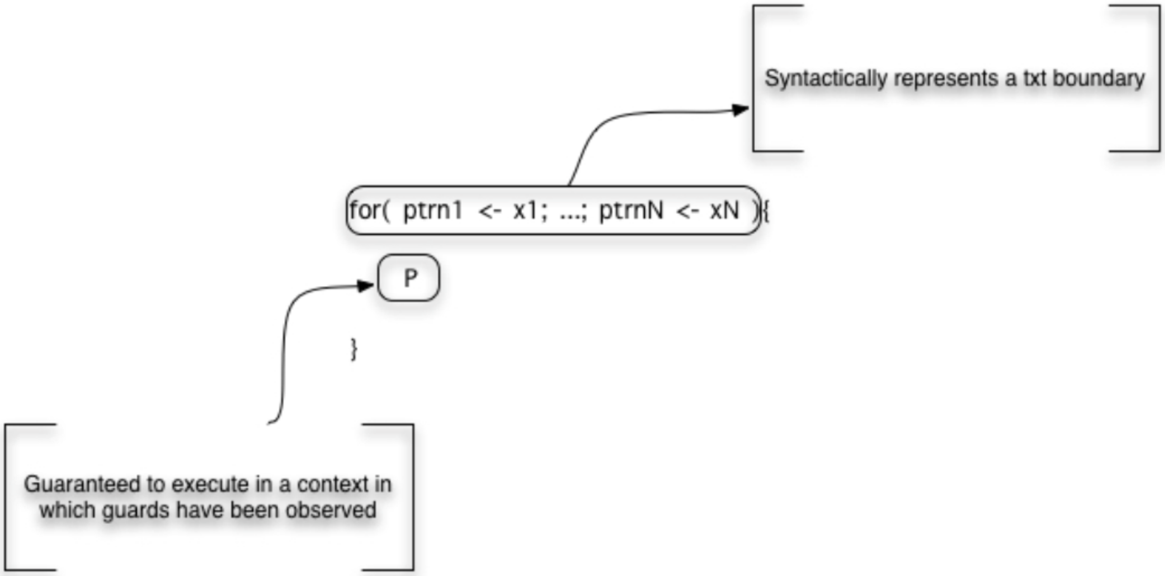

Syntactically represents a txt boundary
for(ptrn1 <- x1; ...; ptrnN <- xN){
P
}
Guaranteed to execute in a context in which guards have been observed

Fig. 3.2: RSON : RChain data representation

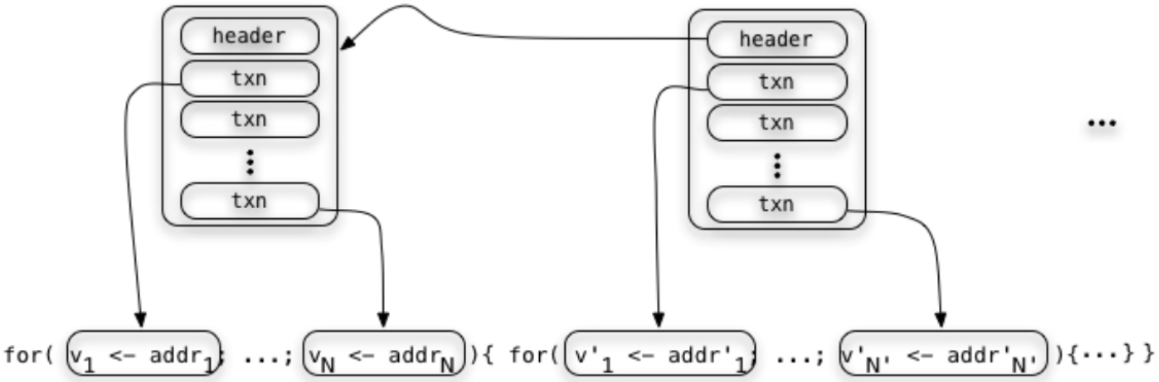

header
txn
txn
txn
header
txn
txn
txn
...
for(v1 <- addr1; ...; vN <- addrN){ for(v'1 <- addr'1; ...; v'N' <- addr'N'){...} }

Fig. 3.3: RSON : RChain data representation
Block representation also embeds directly into RHOCore

The use of a compositional account of namespaces to coordinate amongst the VMs is one of the key ingredients missing in Ethereum's VM design, and the principal reason it is not compositional. The other core change is that the RChain machine design, like the Rosette/ESS design is fundamentally concurrent. Smart contracts in RChain, like actors in Rosette/ESS enjoy fine-grained concurrency during their execution. Two key factors contribute to making this safe for financial transactions.

3.2.1 Concurrency, non-determinism, and safety

The two mechanisms that allow fine-grained concurrent execution to be safe in the distributed setting operate at fundamentally different levels. One is a runtime mechanism and the other is a compile time mechanism. The runtime is easier to understand. The non-determinism arising from concurrent execution associated with a contract always arises as a race of the form:

- two outputs racing to serve one input request

```
x!( v1 ) | for( y <- x )P | x!( v2 )
```

- two input requests vying for a single output

```
for( y <- x )P1 | x!( v ) | for( y <- x )P2
```

Whether that race arises from computation inside the contract or between the contract and its environment. In either of the two possible race cases, for the contract to make progress one of the reductions will be chosen and that choice is the transaction. That's the meaning of the transactional boundary described above. Hence, these are the transactions that are replicated by the Casper consensus algorithm. Thus, while there is internal non-determinism, the replicated state is deterministic. All nodes in the same shard see the same state.

This still makes it possible to write unsafe code. Despite the determinism of the EVM, the DAO bug shows up as a kind of unfairness in scheduling state updates relative to servicing new client requests; and, when expressed as a rholang contract, arises as an unwanted race condition. That is, there is a level of non-determinism that was allowed by the contract that wasn't safe with respect to the intended semantics of the contract. In most practical situations these can be detected and prevented at compile time using the spatial behavioral types of rholang. It is certainly the case in the specific instance of the bug exploited in the attack against the DAO.

3.2.2 What is a VM?

Let's take a moment to review what's in a VM. Every VM corresponds to a table. The table lists a set of transitions. The transitions are of the form:

```
<byte code, machine state> -> <byte code', machine state'>
```

The transitions specify what happens when a machine in a given state encounters a particular byte code instruction:

```
rosette> (code-dump (compile '(+ 1 2)))
litvec:
   0: {RequestExpr}
codevec:
   0: alloc 2
   1: lit 1, arg[0]
   2: lit 2, arg[1]
   3: xfer global[+],trgt
   5: xmit/nxt 2
rosette>
```

Examples include loading a literal into a register or popping register values and adding them. Registers, heap, stack, these are examples of components of the machine state. In the case of the RhoVM the most important transition is the one associated with I/O:

```
for( y <- x )P | x!( Q ) -> P{ @Q / y }
```

This transition says that when an input guarded thread in the VM is waiting for input on x is running concurrently with a thread committing and output on x, then the data passes along x, is bound to the variable y in the continuation P. It is important to understand that this is really a higher level transition that may involve many lower level state changes. This is because x may be bound to a wide variety of channels, from tables in local storage, to AMQP queues, to tcp/ip sockets. Each of these has a natural semantics that interoperates smoothly with this higher level transition rule. The interoperation between this high level transition rule and different channel semantics is precisely what the Tuplespace semantics provides.

What's important for this discussion, however, is the recognition that a given VM instance, i.e. a copy of the VM table plus a specific configuration of machine state, can be restricted to operate on a specific collection of names. This collection of names, what we have been calling a namespace, can be programmatically specified and hence not necessarily finite.

In this architecture a shard corresponds roughly to a namespace and a machine instance and the RChain nodes on in the network on which the state of this VM is stored. We say 'roughly' because shards may be composed of shards, meaning that there are subgroups of the nodes in a given shard that replicate machine state restricted to a subspace of the namespace. Likewise, because VMs can only interoperate if they have overlapping namespaces, multiple shards can be overlaid on the same nodes. This provides both availability and security features because using these facts about the relationships of VM, nodes, and namespaces, finding a correlation between physical locations of nodes and namespaces can be made as computationally hard as desired.

CHAPTER 4

Conclusions and future work

We have provided a brief review of the technology leading up to an effective rendering of the mobile process calculi as a programming model and execution engine for decentralized networks of programmable VMs. We have described a step by step procedure for realizing this on a blockchain in which the state of the VMs are stored.We have provided the background and design rationale for this approach as a correct-by-construction architecture of a blockchain based public, economic, compute infrastructure.

CHAPTER 5

Bibliography

Bibliography

[SM15] Mike Stay and Lucius Gregory Meredith. Higher category models of the pi-calculus. *CoRR*, 2015. URL: http://arxiv.org/abs/1504.04311.

[AH87] Gul Agha and Carl Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. 1987.

[Cai04] Lu'ıs Caires. Behavioral and spatial observations in a logic for the pi-calculus. In *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, 72–89. 2004. URL: http://dx.doi.org/10.1007/978-3-540-24727-2_7, doi:10.1007/978-3-540-24727-2_7.

[FG00] Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, 268–332. 2000. URL: http://dx.doi.org/10.1007/3-540-45699-6_6, doi:10.1007/3-540-45699-6_6.

[GZ97] David Gelernter and Lenore D. Zuck. On what linda is: formal description of linda as a reactive system. In *Coordination Languages and Models, Second International Conference, COORDINATION '97, Berlin, Germany, September 1-3, 1997, Proceedings*, 187–204. 1997. URL: http://dx.doi.org/10.1007/3-540-63383-9_81, doi:10.1007/3-540-63383-9_81.

[HWT14] Jiansen He, Philip Wadler, and Philip W. Trinder. Typecasting actors: from akka to takka. In *Proceedings of the Fifth Annual Scala Workshop, SCALA@ECOOP 2014, Uppsala, Sweden, July 28-29, 2014*, 23–33. 2014. URL: http://doi.acm.org/10.1145/2637647.2637651, doi:10.1145/2637647.2637651.

[KSFS05] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, 192–203. 2005. URL: http://doi.acm.org/10.1145/1086365.1086390, doi:10.1145/1086365.1086390.

[MB03] Greg Meredith and Steve Bjorg. Contracts and types. *Commun. ACM*, 46(10):41–47, 2003. URL: http://doi.acm.org/10.1145/944217.944236, doi:10.1145/944217.944236.

[MR05a] L. Gregory Meredith and Matthias Radestock. A reflective higher-order calculus. *Electr. Notes Theor. Comput. Sci.*, 141(5):49–67, 2005. URL: http://dx.doi.org/10.1016/j.entcs.2005.05.016, doi:10.1016/j.entcs.2005.05.016.

[MR05b] L. Gregory Meredith and Matthias Radestock. Namespace logic: A logic for a reflective higher-order calculus. In *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers*, 353–369. 2005. URL: http://dx.doi.org/10.1007/11580850_19, doi:10.1007/11580850_19.

[Mer15] Lucius Gregory Meredith. Linear types can change the blockchain. *CoRR*, 2015. URL: http://arxiv.org/abs/1506.01001.

[MSD13] Lucius Gregory Meredith, Mike Stay, and Sophia Drossopoulou. Policy as types. *CoRR*, 2013. URL: http://arxiv.org/abs/1307.7766.

[Mil92] Robin Milner. The polyadic pi-calculus (abstract). In *CONCUR '92, Third International Conference on Concurrency Theory, Stony Brook, NY, USA, August 24-27, 1992, Proceedings*, 1. 1992. URL: http://dx.doi.org/10.1007/BFb0084778, doi:10.1007/BFb0084778.

[SMTA95] Munindar P. Singh, Greg Meredith, Christine Tomlinson, and Paul C. Attie. An event algebra for specifying and scheduling workflows. In *Database Systems for Advanced Applications '95, Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA), Singapore, April 11-13, 1995*, 53–60. 1995.

[SM15] Mike Stay and Lucius Gregory Meredith. Higher category models of the pi-calculus. *CoRR*, 2015. URL: http://arxiv.org/abs/1504.04311.

[SM16] Mike Stay and Lucius Gregory Meredith. Logic as a distributive law. *CoRR*, 2016. URL: http://arxiv.org/abs/1610.02247.

[TKS+89] Chris Tomlinson, Won Kim, Mark Scheevel, Vineet Singh, B. Will, and Gul Agha. Rosette: an object-oriented concurrent systems architecture. *SIGPLAN Notices*, 24(4):91–93, 1989. URL: http://doi.acm.org/10.1145/67387.67410, doi:10.1145/67387.67410.

[TCMW93] Christine Tomlinson, Philip Cannata, Greg Meredith, and Darrell Woelk. The extensible services switch in carnot. *IEEE P&DT*, 1(2):16–20, 1993. URL: http://dx.doi.org/10.1109/88.218171, doi:10.1109/88.218171.

[WAC+93] Darrell Woelk, Paul C. Attie, Philip Cannata, Greg Meredith, Amit P. Sheth, Munindar P. Singh, and Christine Tomlinson. Task scheduling using intertask dependencies in carot. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993.*, 491–494. 1993. URL: http://doi.acm.org/10.1145/170035.170150, doi:10.1145/170035.170150.