ATOMIC FAILURE IN WIDE-AREA COMPUTATION

Dominic Duggan
Department of Computer Science
Stevens Institute of Technology
Castle Point on the Hudson
Hoboken, New Jersey 07030, USA.
dduggan@cs.stevens-tech.edu

Abstract The ATF Calculus is a kernel language for wide-area network programming languages, with atomic failure semantics as its central organizing principle.

Keywords: Wide-area network programming, fault tolerance, atomic failure.

1. INTRODUCTION

Programming wide-area networks has emerged as an important area of research, with the exploding commercial interest in WANs for electronic commerce. While programming local-area networks has been a focus of research for many years, the characteristics of WANs emphasize different issues and potentially require different solutions. One fundamental difference is that, for security reasons, WANs are partitioned by firewalls into administrative domains. Any application that spans such administrative domains must somehow navigate the firewalls that enforce the partitioning. This difference is so fundamental that it has been argued that it calls for a wholly new model of computation for programming WANs [7]. Although firewalls are to some extent becoming obsolete because of encrypted traffic, this argument for partitioned network address spaces still applies in the emerging environment of virtual private networks.

Several process algebras have been developed as the basis for new programming languages for wide-area applications. At least two of these, the Ambient Calculus [8] and the Seal Calculus [10], are based on the idea that networks are partitioned into administrative domains and applications must navigate between these domains. These algebras have *mobile computation* as their organizing principle: to cope with the problems with communication in WANs, they allow processes to migrate across the network, allowing an application to move to a host where it needs to perform some computation. A central facet in both of these calculi is their support for navigating firewalls, one of the central motivations for wide-area network programming languages [7].

This paper introduces the ATF Calculus, intended as a kernel language for wide-area programming languages. Although taking some of its cue from the aforementioned calculi, the ATF Calculus takes *atomic failure* as its main organizing principle. The motivation for this is that typical wide-area applications require some notion of coordination between geographically and administratively distributed sites. The distribution of the sites raises the possibility of localized site failures and network partitions, and there should be some support for coping with such failures. In an application involving coordination between several sites, it should be possible to commit or abort a computation based on the pattern of failures of the sites involved in the computation.

Transaction systems [19], and in particular nested transactions [21, 24], have been developed for providing the capability for atomic failures in databases and distributed systems. The ATF Calculus does not completely follow the semantics of traditional transaction systems. In particular concurrency control is considered as an application issue, and the ATF Calculus only considers support for making failure atomic. This support is based on two ingredients:

- Tracking of causal dependencies between computations.
- An operation for atomic commitment of a collection of transactions.

In Sect. 2 we give a description of the network model underlying the ATF Calculus. In Sect. 3 we give a description of the process calculus describing programs that execute within transactions. Both of these descriptions include type systems for well-formed networks and transactions. Sect. 4 gives an operational semantics for the calculus. Sect. 5 gives a semantics for aborting and committing transactions. Sect. 6 considers an extension of the calculus with process mobility. Finally Sect. 7 considers related work and provides conclusions.

2. NETWORK MODEL

Traditional process calculi such as CCS [22] and the π-calculus [23] do not distinguish between processes and the medium through which they communicate; the medium is represented as just another process. The ambient calculus [8] does not distinguish between networks, hosts and process address spaces; all are represented as ambients. We find it useful to distinguish between places on the network (hosts and subnets) and processes, because of their different failure characteristics. In this section we describe our model of networks. In the next section we describe the processes that execute over these networks.

Networks in the ATF Calculus are described as follows:

$n \in$ Name	$::=$	n^{pl}	Place Name
	$\mid$	n^{ch}	Channel Name
	$\mid$	n^{pk}	Packet Name
	$\mid$	t	Transaction Name
$N \in$ Network	$::=$	$n^{\text{pl}}[\mathcal{L}, N]$	Place
	$\mid$	$t\{P\}$	Process
	$\mid$	$(N_1 \mid N_2)$	Wire
	$\mid$	$(new\ n : NT)N$	New Name
	$\mid$	$\mathbf{0}$	Empty Network
$\mathcal{L} \in$ Log	$::=$	ε	Empty Log
	$\mid$	N	Log Entry
	$\mid$	$(\mathcal{L}_1 ; \mathcal{L}_2)$	Log Extension

A place is an administrative domain or a host on the network. This contains a "soup" of processes, channels and subdomains N. n^{pl} is the name of the place, while $\mathcal{L}$ is a log of the actions performed at that place. For fault tolerance this log must be kept in stable storage.

A channel is used for asynchronous communication between processes. As in the Ambient Calculus and the Seal Calculus, all communication is local within an administrative domain. To communicate with a remote host, there must be an application-level protocol for delivering a message across the intervening domains. As with places, a channel has a name n^{ch}. The contents of a channel n^{ch} at a place n^{pl} is obtained by taking the union of the atoms representing messages sent to that channel at that place. Messages are a particular instance of transactional processes.

A *transactional process* $t\{P\}$ denotes a process P that executes as part of the transaction identified by the transaction identifier t. If that transaction is aborted, then all effects (messages sent and received) of that process must be undone. Transactional processes $t\{P\}$ reflect the fact that all application-level computations in the ATF Calculus take place within a transaction. The transaction name t identifies the transaction within which a process executes ($t\{P\}$). Multiple threads may execute within a transaction; if one thread aborts in a transaction, then all effects of all threads in that transaction are undone. Thus transactions are the mechanism for defining the granularity of failures in the ATF Calculus.

2.1. NETWORK TYPE RULES FOR ATF

The type rules for the network are described by the following type rules, using judgements of the form $\Gamma \vdash N$ **net** where Γ is a context of named types $(n : NT)$.

$$\frac{\Gamma \vdash N\ \mathbf{net} \quad \Gamma \vdash \mathcal{L}\ \mathbf{log} \quad (n^{\mathrm{pl}} : \mathit{Place}) \in \Gamma}{\Gamma \vdash n^{\mathrm{pl}}[\mathcal{L}, N]\ \mathbf{net}} \qquad (\textsc{Net Place})$$

$$\frac{\Gamma \vdash N_1\ \mathbf{net} \quad \Gamma \vdash N_2\ \mathbf{net}}{\Gamma \vdash (N_1 \mid N_2)\ \mathbf{net}} \qquad (\textsc{Net Wire})$$

$$\frac{\Gamma \cup \{(n : NT)\} \vdash N\ \mathbf{net}}{\Gamma \vdash (\mathit{new}\ n : NT) N\ \mathbf{net}} \qquad (\textsc{Net New})$$

$$\frac{(t : \mathit{Trans}) \in \Gamma \quad \Gamma \vdash P : ()}{\Gamma \vdash t\{P\}\ \mathbf{net}} \qquad (\textsc{Net Process})$$

$$\frac{\Gamma \vdash N\ \mathbf{net}}{\Gamma \vdash N\ \mathbf{log}} \qquad (\textsc{Log Net})$$

$$\frac{\Gamma \vdash \mathcal{L}_1\ \mathbf{log} \quad \Gamma \vdash \mathcal{L}_2\ \mathbf{log}}{\Gamma \vdash \mathcal{L}_1 ; \mathcal{L}_2\ \mathbf{log}} \qquad (\textsc{Log Extend})$$

There is a further side-condition, that we do not explicitly enforce here for economy: All transactional processes for a particular transaction t must execute at the same place. This is because various decisions regarding committing and aborting transactions are made based on information in the logs, and so for scalability it must be possible to obtain this log information at a single local place. Note that it is not sufficient to say that there is a mapping from a transaction identifier t to the name of the place n^{pl} on which all processes in that transaction execute. This is insufficient because we do not require that every place has a unique name (for example, network address translation in VPNs would make this an unrealistic assumption).

3. TRANSACTIONAL PROCESSES

Processes in the ATF Calculus are described by:

$P \in$ Process	::=	$\mathit{send}(M, V)$	Message Send
	$\mid$	$\mathit{receive}(M, F, P)$	Message Receive

	$\mid$	$crypt(M,V,F)$	Packet Encrypt
	$\mid$	$decrypt(M,V,F)$	Packet Decrypt
	$\mid$	$F(V)$	Continue
	$\mid$	$let\ \langle \overline{x_n} \rangle = V\ in\ P$	Elim Tuple
	$\mid$	$!P$	Replication
	$\mid$	$(P_1 \mid P_2)$	Parallel Composition
	$\mid$	$(new\ n : NT)P$	New Name
	$\mid$	*commit*	Commit transaction
	$\mid$	*abort*	Abort transaction
	$\mid$	$\mathbf{0}$	Stopped Process
$F \in$ Cont	$::=$	$(x : T)P$	Continuation

The basic operations are for asynchronous message-passing [27]. In the ATF Calculus the essential use of mobility for navigating administrative domains is in the *send* and *receive* operations. For example the *send* operation takes two arguments: a capability M and a value V. The capability specifies a path to be taken in the network (identified by subcapabilities for leaving and entering administrative domains) and a capability for depositing the value in a channel at the final destination place. As with algebras such as the Ambient Calculus and the Spi Calculus [8, 1], access control is enforced by controlling the distribution of these capabilities, which are akin to private keys in cryptographic infrastructures.

Our provision for "mobility" is consistent with approaches in active networks, such as the Switchware architecture [2], that restrict mobile threads to simplified packet languages (such as the PLAN language of the Switchware architecture [18]). The language of capabilities M can be considered as the analogue in the ATF Calculus of packet languages such as PLAN. This is in contrast with the Ambient Calculus and the Seal Calculus, which allow general user processes to migrate across the network. Active network architectures also allow heavyweight loading of application code modules to routers; this could be accomplished in the ATF Calculus by extending it to allow processes in messages. This is investigated in Sect. 6.

The *receive* operation takes as its main argument a capability for reading from a channel. This operation also has two continuations: the success continuation F and the failure continuation P, where the latter is activated if the receive operation times out. We assume an asynchronous system, so timeouts provide a form of unreliable failure detector which is the best that we can attain in an asynchronous system.

Values in the ATF Calculus are described by:

$M, V \in$ Value	$::=$	$n^{\mathrm{pk}}[V]$	Packet
	$\mid$	$\langle V_1, \ldots, V_k \rangle$	Tuple
	$\mid$	p	Parameter
	$\mid$	$\mathit{in}\ p$	Input Capability
	$\mid$	$\mathit{out}\ p$	Output Capability
	$\mid$	$M_1.M_2$	Compose Caps

Packets are an inessential aspect of the ATF Calculus, but provide cryptographic primitives for authentication and encryption of message contents as part of the language. Our motivation for including packets is that they are a fundamental tool for WAN programming, provided as primitives in the Spi-calculus [1] and defined in the ambient calculus [9]. The design of the ATF Calculus was influenced by the need to provide a corresponding facility.

Creating a packet requires a capability for putting a value into a packet, while reading a packet requires the inverse capability. Capabilities therefore provide a function analogous to cryptographic keys; an application may publish only a key for creating packets, and then be the only process capable of reading packets created using the capability. Separating the capabilities for message creation and reading, from the capabilities for delivery and receipt of messages, allows an encrypted message to be forwarded by a process that does not have access to a key for reading the message contents.

The form of capabilities are taken from the Ambient Calculus. In the Ambient Calculus, places, channels and packets are uniformly represented as ambients. However we find it easier to treat each of these concepts differently. For fault tolerance purposes, the behaviour of places, channels and packets are very different. A capability is a sequence of subcapabilities of the form *in p* or *out p*, where p is a place name, channel name or packet name, with the interpretation:

$\mathit{in}\ n^{\mathrm{pl}}$	enter a place
$\mathit{in}\ n^{\mathrm{ch}}$	write to a channel
$\mathit{in}\ n^{\mathrm{pk}}$	create a packet

$\mathit{out}\ n^{\mathrm{pl}}$	leave a place
$\mathit{out}\ n^{\mathrm{ch}}$	read a channel
$\mathit{out}\ n^{\mathrm{pk}}$	read a packet

Types in the ATF Calculus are described by:

$AT \in$ Ambient Type	$::=$	$Place$	Place Type
	$\mid$	$Chan[T]$	Channel Type
	$\mid$	$Packet[T]$	Packet Type
$NT \in$ Name Type	$::=$	AT	Ambient Type
	$\mid$	$Trans$	Transaction Type
$T \in$ Type	$::=$	NT	Name Type
	$\mid$	$\langle T_1, \ldots, T_k \rangle$	Tuple Type
	$\mid$	$Cap[AT]$	Capability Type
	$\mid$	X	Type Variable
	$\mid$	$\mu X.T$	Recursive Type
Continuation Type	$::=$	$T \to ()$	Process Type

Ambient types are the types of place names, channel names and packet names (the name reflects the original inspiration from the Ambient Calculus). Types also include process types and capability types, where the latter are indexed by ambient types. Types also include recursive types. In combination with tuple types, this supports a polyadic sorting discipline for data structures such as described for the π-calculus [23].

3.1. PROCESS TYPE RULES FOR ATF

In this subsection we consider the type rules for transactional processes.

$$\frac{\Gamma \vdash M : Cap[Chan[T]] \quad \Gamma \vdash V : T}{\Gamma \vdash send(M,V) : ()} \quad \text{(PROC SEND)}$$

As already described, the send operation has two arguments, a capability and a value. The capability is to allow the process to deposit a message payload into a channel of the same type as the type of the payload. The send operation generates an "active message" that navigates the network to its destination.

$$\frac{\Gamma \vdash M : Cap[Chan[T]] \quad \Gamma \vdash F : T \to () \quad \Gamma \vdash P : ()}{\Gamma \vdash receive(M,F,P) : ()} \quad \text{(PROC RECEIVE)}$$

The receive operation requires a capability to read the contents of a message channel, where the message payload has the same type as the domain of the success continuation F. As with the typed ambient calculus [9], the type system does not distinguish between capabilities for sending to a channel and for receiving from a channel.

$$\frac{\Gamma \vdash M : Cap[Packet[T]] \quad \Gamma \vdash V : T \quad \Gamma \vdash F : Packet[T] \to ()}{\Gamma \vdash crypt(M^{net}, V, F) : ()}$$

(PROC ENCRYPT)

$$\frac{\Gamma \vdash M : Cap[Packet[T]] \quad \Gamma \vdash V : Packet[T] \quad \Gamma \vdash F : T \to ()}{\Gamma \vdash decrypt(M^{net}, V, F) : ()}$$

(PROC DECRYPT)

The encryption operation takes a capability for creating a packet and a payload, and passes to the continuation an encrypted packet containing the payload. Conversely the decryption operation takes a capability for destructing a packet, and an encrypted packet, and passes the contents of the packet to the continuation. Again the type system does not distinguish between capabilities for creating and for destructing packets.

The type rules for the remaining process constructs are fairly conventional:

$$\frac{\Gamma \vdash F : T \to () \quad \Gamma \vdash V : T}{\Gamma \vdash F(V) : ()}$$ (PROC CONTINUE)

$$\frac{\Gamma \vdash V : \langle T_1, \ldots, T_k \rangle \quad \Gamma \cup \{(x_1 : T_1), \ldots, (x_k : T_k)\} \vdash P : ()}{\Gamma \vdash let\ \langle x_1, \ldots, x_k \rangle = V\ in\ P : ()}$$ (PROC LET)

$$\frac{\Gamma \vdash P : ()}{\Gamma \vdash !P : ()}$$ (PROC REPLICATE)

$$\frac{\Gamma \cup \{(n : NT)\} \vdash P : ()}{\Gamma \vdash (new\ n : NT)P : ()}$$ (PROC NEW)

$$\frac{\Gamma \vdash P_1 : () \quad \Gamma \vdash P_2 : ()}{\Gamma \vdash (P_1 \mid P_2) : ()}$$ (PROC PAR)

$$\Gamma \vdash \mathbf{0} : ()$$ (PROC NULL)

$$\frac{\Gamma \cup \{(x : T)\} \vdash P : ()}{\Gamma \vdash ((x : T)P) : T \to ()}$$ (PROC CONT)

The PROC CONT rule provides the type rule for a continuation, representing the remainder of a computation. As with almost all process algebras, programs are written in continuation-passing style (a process algebra is basically an assembly language for concurrent programming). Continuations are second-class; they cannot be treated as values.

The calculus is first-order, in the sense that only simple values can be exchanged between processes. The calculus could be made higher-order by making continuations into first-class values, essentially adding λ-abstraction to the language. This is a straightforward modification to the operational semantics, but at some cost in complicating the metatheory of the calculus. We consider another approach to adding process mobility in Sect. 6.

3.2. VALUE TYPE RULES FOR ATF

In this subsection we provide the type rules for values, that are the content of message payloads exchanged between processes.

$$\frac{(n^{\mathrm{pk}} : \mathit{Packet}[T]) \in \Gamma \quad \Gamma \vdash V : T}{\Gamma \vdash n^{\mathrm{pk}}[V] : \mathit{Packet}[T]} \qquad \text{(VAL PACKET)}$$

$$\frac{\Gamma \vdash V_1 : T_1 \quad \ldots \quad \Gamma \vdash V_k : T_k}{\Gamma \vdash \langle V_1, \ldots, V_k \rangle : \langle T_1, \ldots, T_k \rangle} \qquad \text{(VAL TUPLE)}$$

$$\frac{(p : T) \in \Gamma}{\Gamma \vdash p : T} \qquad \text{(VAL PARAM)}$$

The VAL PARAM rule is for both names (bound by the *new* construct) and variables (bound by abstraction $((x : T)P)$). A parameter can be a variable abstracting over a place, channel or packet name, as reflected in the rules for forming capabilities.

$$\frac{\Gamma \vdash p : AT}{\Gamma \vdash \mathit{in}\ p : \mathit{Cap}[AT]} \qquad \text{(VAL INPUT CAP)}$$

$$\frac{\Gamma \vdash p : AT}{\Gamma \vdash \mathit{out}\ p : \mathit{Cap}[AT]} \qquad \text{(VAL OUTPUT CAP)}$$

$$\frac{\Gamma \vdash M_1 : \mathit{Cap}[\mathit{Place}] \quad \Gamma \vdash M_2 : \mathit{Cap}[AT]}{\Gamma \vdash M_1.M_2 : \mathit{Cap}[AT]} \qquad \text{(VAL COMPOSE CAP)}$$

The VAL COMPOSE CAP rule types the composition of capabilities. As explanation for this type rule, useful capabilities have the following forms:

$\mathit{in}\ n^{\mathrm{pk}}, \mathit{out}\ n^{\mathrm{pk}}$	Create or destruct a packet
$\mathit{out}\ n^{\mathrm{ch}}$	Receive from local channel
$M_1 \ldots M_n.\mathit{in}\ n^{\mathrm{ch}}$	Deliver payload to channel at destination

where in the latter case each M_i has the form *in* n_i^{pl} or *out* n_i^{pl} for some n_i^{pl}, $i = 1, \ldots, n$.

4. OPERATIONAL SEMANTICS

This section provides the operational semantics for the ATF Calculus. In Sect. 5 we consider the semantics of committing and aborting transactions.

4.1. LOCAL COMPUTATION RULES

The local computation rules are fairly standard, and describe sequential computation within a process. The local computation rules are given by:

$$!P \rightarrow (!P \mid P) \qquad \text{(RED REPLICATE)}$$

$$\mathit{receive}(\mathit{out}\; n^{\text{ch}}, F, P) \rightarrow P \qquad \text{(RED TIMEOUT)}$$

$$\mathit{crypt}(\mathit{in}\; n^{\text{pk}}, V, F) \rightarrow F(n^{\text{pk}}[V]) \qquad \text{(RED CRYPT)}$$

$$\mathit{decrypt}(\mathit{out}\; n^{\text{pk}}, n^{\text{pk}}[V], F) \rightarrow F(V) \qquad \text{(RED DECRYPT)}$$

$$\mathit{let}\; \langle x_1, \ldots, x_k \rangle = \langle V_1, \ldots, V_k \rangle\; \mathit{in}\; P \rightarrow \{V_1/x_1, \ldots, V_n/x_n\}P \qquad \text{(RED LET)}$$

$$((x:T)P)(V) \rightarrow \{V/x\}P \qquad \text{(RED APP)}$$

The timeout rule allows the receive operation to time out after some period of time. We do not explicitly represent time, since it does not contribute anything to the semantics. In a synchronous system it might be useful to have a representation of time in the semantics, although calculi that assume the fail-stop model of failures only rely on an operation for detecting whether a site has failed [25, 13]. As is well-known, wide-area networks constitute asynchronous distributed systems, where no upper bound can be placed on message transmission delays or message processing times, and there may be unbounded real-time clock drift between different machines [11]. Timeouts therefore constitute unreliable failure detectors where a process makes the assumption after some period of time that a failure has occurred. The process must be prepared to cope with the possibility that this assumption is erroneous.

4.2. STRUCTURAL RULES

The structural rules for networks and processes follow the usual style of "chemical" semantics for concurrent processes [6], and constitute the "heating" rules that allow processes and messages to be brought together for synchronization. The structural rules for processes include the internal reduction relation for processes. We omit the obvious structural congruence rules.

$$\mathbf{0} \mid N \equiv N$$

$$N_1 \mid N_2 \equiv N_2 \mid N_1$$

$$(N_1 \mid N_2) \mid N_3 \equiv N_1 \mid (N_2 \mid N_3)$$

$$((\mathit{new}\ n{:}NT)N_1) \mid N_2 \equiv (\mathit{new}\ n{:}NT)(N_1 \mid N_2),\ \ n \notin \mathit{fn}(N_2)$$

$$n^{\mathrm{pl}}[\mathcal{L}, (\mathit{new}\ n{:}NT)N] \equiv (\mathit{new}\ n{:}NT)n^{\mathrm{pl}}[\mathcal{L}, N]$$

$$(\mathit{new}\ n_1{:}NT)(\mathit{new}\ n_2{:}NT)N \equiv (\mathit{new}\ n_2{:}NT)(\mathit{new}\ n_1{:}NT)N,\ \ n_1 \neq n_2$$

$$(\mathit{new}\ n{:}NT)N \equiv N,\ \ n \notin \mathit{fn}(N)$$

$$t\{P_1 \mid P_2\} \equiv (t\{P_1\} \mid t\{P_2\})$$

$$t\{(\mathit{new}\ n{:}NT)P\} \equiv (\mathit{new}\ n{:}NT)t\{P\}$$

$$\varepsilon; \mathcal{L} \equiv \mathcal{L}$$

$$\mathcal{L}; \varepsilon \equiv \mathcal{L}$$

$$\mathcal{L}_1; (\mathcal{L}_2; \mathcal{L}_3) \equiv (\mathcal{L}_1; \mathcal{L}_2); \mathcal{L}_3$$

$$\mathbf{0} \mid P \equiv P$$

$$P_1 \mid P_2 \equiv P_2 \mid P_1$$

$$(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$$

$$((\mathit{new}\ n{:}NT)P_1) \mid P_2 \equiv (\mathit{new}\ n{:}NT)(P_1 \mid P_2),\ \ n \notin \mathit{fn}(P_2)$$

$$(\mathit{new}\ n_1{:}NT)(\mathit{new}\ n_2{:}NT)P \equiv (\mathit{new}\ n_2{:}NT)(\mathit{new}\ n_1{:}NT)P,\ \ n_1 \neq n_2$$

$$(\mathit{new}\ n{:}NT)P \equiv P,\ \ n \notin \mathit{fn}(P)$$

$$\varepsilon.M \equiv M$$

$$M.\varepsilon \equiv M$$

$$(M_1.M_2).M_3 \equiv M_1.(M_2.M_3)$$

$$\frac{P_1 \rightarrow P_2}{P_1 \equiv P_2}$$

4.3. NETWORK AND LOG PREDICATES

Several of the computation rules must examine the logs in order to check preconditions for their successful execution. The predicates that they evaluate are given by the rules below. The predicates check if a transaction has successfully committed, if a transaction has aborted, if a transaction has received a particular message, and if a transaction depends causally on another transaction. Causal dependencies are introduced between transactions t_1 and t_2 when t_2 receives a message from t_1.

$$\frac{\mathcal{L}_i \models \mathcal{J}}{(\mathcal{L}_1;\mathcal{L}_2) \models \mathcal{J}} \quad \text{(Log Predicate Extend)}$$

$$t\{commit\} \models t \textbf{ committed} \quad \text{(Log Predicate Committed)}$$

$$t\{abort\} \models t \textbf{ aborted} \quad \text{(Log Predicate Aborted)}$$

$$\frac{N^{\mathrm{recv}} \equiv t_1\{receive(out\; n^{\mathrm{ch}}, F, P)\} \quad N^{\mathrm{mesg}} \equiv t_2\{send(in\; n^{\mathrm{ch}}, V)\}}{(N^{\mathrm{recv}} \mid N^{\mathrm{mesg}}) \models t_1 \textbf{ received } N^{\mathrm{mesg}}} \quad \text{(Log Predicate Recvd)}$$

$$\frac{\mathcal{L} \models t_1 \textbf{ received } N^{\mathrm{mesg}} \qquad N^{\mathrm{mesg}} \equiv t_2\{send(in\; n^{\mathrm{ch}}, V)\}}{\mathcal{L} \models t_2 \rightsquigarrow t_1} \quad \text{(Log Predicate Depends)}$$

This last rule determines if there is a causal dependency from a process in the transaction t_2 to a process in the transaction t_1. If there is such a dependency, then if the transaction t_2 aborts, the transaction t_1 is also required to abort. We extend these log predicates $\mathcal{L} \models \mathcal{J}$ to to network predicates $N \models \mathcal{J}$:

$$\frac{N_i \models \mathcal{J}}{(N_1 \mid N_2) \models \mathcal{J}} \quad \text{(Net Predicate Par)}$$

$$\frac{n \notin fn(\mathcal{J}) \quad N \models \mathcal{J}}{(new\; n : NT)N \models \mathcal{J}} \quad \text{(Net Predicate New)}$$

$$\frac{\mathcal{L} \models \mathcal{J} \text{ or } N \models \mathcal{J}}{n^{\mathrm{pl}}[\mathcal{L}, N] \models \mathcal{J}} \quad \text{(Net Predicate Place)}$$

4.4. NETWORK NAVIGATION RULES

The semantics of the ATF Calculus is based on messages that navigate administrative domain boundaries and eventually "deposit themselves" in channels at the destination place. This message can then only be read by a process executing the *receive* operation at that place, so message-passing is strictly local. All of this is similar to algebras for mobile computation such as the Ambient Calculus and the Seal Calculus. The following rules allow a message to navigate in and out of administrative domains, with the crossing of firewalls mediated by capabilities associated with the message:

$$\frac{N' \equiv t\{send(M,V)\}}{t\{send(in\, n^{\mathrm{pl}}.M,V)\} \mid n^{\mathrm{pl}}[\mathcal{L},N] \longrightarrow n^{\mathrm{pl}}[\mathcal{L},(N' \mid N)]} \quad (\textsc{Red In Place})$$

$$\frac{N' \equiv t\{send(M,V)\}}{n^{\mathrm{pl}}[\mathcal{L},(t\{send(out\, n^{\mathrm{pl}}.M,V)\} \mid N)] \longrightarrow (N' \mid n^{\mathrm{pl}}[\mathcal{L},N])}$$
$$(\textsc{Red Out Place})$$

4.5. COMMUNICATION RULE

The Red Recv rule allows allows a message to be retrieved from a channel by a receiving process. This is the synchronization point in the calculus. To a large extent it is the most important reduction rule in the calculus, since it establishes a causal relationship between the sending and receiving transaction. A log entry is written to stable storage, both to record a change in the state of the place, and also to record the causal dependency introduced by synchronization in the log:

$$\frac{\begin{array}{c} N^{\mathrm{recv}} \equiv t_1\{receive(out\, n^{\mathrm{ch}},F,P)\} \\ N^{\mathrm{mesg}} \equiv t_2\{send(in\, n^{\mathrm{ch}},V)\} \quad N' \equiv t_1\{F(V)\} \\ \mathcal{L} \not\models t_1\ \textbf{aborted} \quad \mathcal{L} \not\models t_2\ \textbf{committed} \quad \mathcal{L}' \equiv \mathcal{L};(N^{\mathrm{recv}} \mid N^{\mathrm{mesg}}) \end{array}}{n^{\mathrm{pl}}[\mathcal{L},(N^{\mathrm{recv}} \mid N^{\mathrm{mesg}} \mid N)] \longrightarrow n^{\mathrm{pl}}[\mathcal{L}',(N' \mid N)]}$$
$$(\textsc{Red Recv})$$

The side-condition that $\mathcal{L} \not\models t_2$ **committed** reflects that a committed process cannot receive any new messages, since it might introduce causal dependencies on uncommitted transactions that subsequently aborted.

5. COMMIT AND ABORT

In this section we consider the semantics of the commit and abort operations. The abort operation writes an entry on the log and rolls back the computation of the aborted process. "Rollback" in this respect simply entails restoring messages that were received by this process.

$$\frac{\mathcal{L} \not\models t\ \textbf{committed} \quad \mathcal{L} \not\models t\ \textbf{aborted} \quad \{N_1,\ldots,N_k\} = \{N \mid \mathcal{L} \models t\ \textbf{received}\ N\} \quad N' \equiv \mathcal{K}[\![(N \mid (N_1 \mid \ldots \mid N_k))]\!]t}{n^{\mathrm{pl}}[\mathcal{L},(N \mid t\{abort\})] \longrightarrow n^{\mathrm{pl}}[(\mathcal{L};t\{abort\}),N']} \quad \text{(Red Abort)}$$

The metafunction $\mathcal{K}[\![N]\!]t$ denotes the process of killing any processes in the transaction t:

$$\begin{array}{rcl} \mathcal{K}[\![n^{\mathrm{pl}}[\mathcal{L},N]]\!]t & = & n^{\mathrm{pl}}[\mathcal{L},N] \\ \mathcal{K}[\![N_1 \mid N_2]\!]t & = & (\mathcal{K}[\![N_1]\!]t \mid \mathcal{K}[\![N_2]\!]t) \\ \mathcal{K}[\![(new\ n : NT)N]\!]t & = & (new\ n : NT)\,\mathcal{K}[\![N]\!]t \\ \mathcal{K}[\![t'\{P\}]\!]t & = & \begin{cases} t'\{P\} & \text{if } t \neq t' \\ \mathbf{0} & \text{otherwise} \end{cases} \end{array}$$

For simplicity in this account, we do not roll back messages sent by an aborted process (once they have left the local site). Any process that receives such a message will be unable to commit. For local processes, the Red Recv rule ensures that such processes ignore messages sent by processes that execute within transactions locally to this place (such a message may return after wandering around the network). However it is unclear how to achieve a scalable version of this for processes that may receive messages from an aborted remote transaction.

It remains to give the semantics of the *commit* operation. The commitability predicate plays an important role in defining the preconditions for this operation. A transaction t_1 is allowed to commit if all transactions t_2 that it depends on causally have committed, or are trying to commit. If the latter case t_1 does commit, then t_2 must also commit. The complication with commitment is that a collection of transactions may be causally dependent on each other. We do not attempt to prevent causal cycles because there is no scalable way to enforce such a restriction. Instead several transactions that are in a causal cycle must commit simultaneously. The commitability predicate determines that a collection of transactions $\{t_1,\ldots,t_k\}$ are ready to commit.

Definition 1 (Commitability) *The (possibly mutually dependent) transactions $t_1,\ldots,t_k$ can be committed in the network N, written $N \models \{t_1,\ldots,t_k\}$* **commitable**, *if it can be justified by the following inference rules:*

$$\frac{N_1 \models \{t_1,\ldots,t_k\}\ \textbf{commitable} \quad N_2 \models \{t_1,\ldots,t_k\}\ \textbf{commitable}}{(N_1 \mid N_2) \models \{t_1,\ldots,t_k\}\ \textbf{commitable}}$$

(Commit Wire)

$$\frac{NT \neq \mathit{Trans} \quad N \models \{t_1,\ldots,t_k\}\ \textbf{commitable}}{(\mathit{new}\ n : NT)N \models \{t_1,\ldots,t_k\}\ \textbf{commitable}} \quad \text{(COMMIT NEW)}$$

$$\frac{t \notin \{t_1,\ldots,t_k\} \quad N \models t\ \textbf{committed}\ \text{or}\ (N \not\models t \leadsto t_i\ \text{for}\ i = 1,\ldots,k) \quad N \models \{t_1,\ldots,t_k\}\ \textbf{commitable}}{(\mathit{new}\ t : \mathit{Trans})N \models \{t_1,\ldots,t_k\}\ \textbf{commitable}}$$

(COMMIT NEW TRANS)

$$\frac{\mathcal{L} \not\models t_i\ \textbf{aborted}\ \text{and}\ \mathcal{L} \not\models t_i\ \textbf{committed}\ \text{for}\ i = 1,\ldots,k \quad N \models \{t_1,\ldots,t_k\}\ \textbf{commitable}}{n^{\mathrm{pl}}[\mathcal{L},N] \models \{t_1,\ldots,t_k\}\ \textbf{commitable}} \quad \text{(COMMIT PLACE)}$$

$$t\{P\} \models \{t_1,\ldots,t_k\}\ \textbf{commitable} \quad \text{(COMMIT PROC)}$$

$$\mathbf{0} \models \{t_1,\ldots,t_k\}\ \textbf{commitable} \quad \text{(COMMIT NULL)}$$

$$\frac{N' \models \{t_1,\ldots,t_k\}\ \textbf{commitable} \quad N' \equiv N}{N \models \{t_1,\ldots,t_k\}\ \textbf{commitable}} \quad \text{(COMMIT CONG)}$$

Define network contexts and commit contexts by:

$$\begin{array}{rcl}
\mathrm{N}[\,] & ::= & [\,] \mid n^{\mathrm{pl}}[\mathcal{L},\mathrm{N}[\,]] \mid (N \mid \mathrm{N}[\,]) \mid (\mathit{new}\ n : NT)\mathrm{N}[\,] \\
\mathrm{C}[X_1,\ldots,X_m] & ::= & \mathrm{N}[X_1] \\
 & \mid & n^{\mathrm{pl}}[\mathcal{L},\mathrm{C}[X_1,\ldots,X_m]] \\
 & \mid & (N \mid \mathrm{C}[X_1,\ldots,X_m]) \\
 & \mid & (\mathit{new}\ n : NT)\mathrm{C}[X_1,\ldots,X_m] \\
 & \mid & (\mathrm{C}[X_1,\ldots,X_k] \mid \mathrm{C}[X_{k+1},\ldots,X_m])
\end{array}$$

where, for commit contexts, in the last case $0 \leq k \leq m$, and in the first case $m = 1$. A commit context is used to describe the simultaneous atomic commitment of k transactions. Simultaneous commitment is necessary because of the possibility of causal cycles in the causal dependency graph.

Definition 2 (Global Computation) *The global computation relation $N_1 \longrightarrow N_2$ relates networks N_1 and N_2 according to the following rules:*

$$\frac{N_1 \equiv N_1' \quad N_1' \longrightarrow N_2' \quad N_2' \equiv N_2}{N_1 \longrightarrow N_2} \quad \text{(RED CONG)}$$

$$\frac{N_1 \longrightarrow N_2}{\mathrm{N}[N_1] \longrightarrow \mathrm{N}[N_2]} \quad \text{(RED NET)}$$

$$\frac{P_1 \longrightarrow P_2}{\mathrm{N}[t\{P_1\}] \longrightarrow \mathrm{N}[t\{P_2\}]} \qquad \text{(RED PROC)}$$

$$\frac{N \equiv \mathrm{C}[N_1,\ldots,N_k] \quad N \models \{t_1,\ldots,t_k\}\ \textbf{commitable} \qquad \left\{ \begin{array}{l} N_i \equiv n_i^{\mathrm{pl}}[\mathcal{L}_i,(N_i' \mid t_i\{commit\})] \\ N_i'' \equiv n_i^{\mathrm{pl}}[(\mathcal{L}_i;t_i\{commit\}),N_i'] \end{array} \right\} \text{for } i=1,\ldots,k}{(new\ \overline{t_k} : Trans)\mathrm{C}[N_1,\ldots,N_k] \longrightarrow (new\ \overline{t_k} : Trans)\mathrm{C}[N_1'',\ldots,N_k'']}$$
(RED COMMIT)

We rely on a protocol such as the distributed two-phase commit protocol to actually implement the commit operation [19]. It is well-known that it is not possible to devise a non-blocking protocol for atomic commitment in an asynchronous system [16, 14].

Our intent is to describe and motivate the calculus, and so we eschew detailed discussion of its safety properties. The obvious safety property that we can check is type safety:

Lemma 1 *Let* $\overset{*}{\longrightarrow}$ *denote the reflexive transitive closure of* $\longrightarrow$. *If* $\Gamma \vdash N$ **net** *and* $N \overset{*}{\longrightarrow} N'$*, then* $\Gamma \vdash N'$ **net**.

We can verify other safety properties of the calculus by giving a translation to a semantics where the effects of aborted transactions are undone. Messages originating in such transactions are removed from message channels, and the records of the effects of such transactions are erased from logs. This provides a verification of the calculus in the sense that any effect obtained from a "run" of the network, and remaining after transactions have aborted, could also be obtained from a run where no aborted transaction ran in the first place. This is the *causal consistency* property of the calculus: the commit and abort rules ensure that any effects remaining after undoing the effects of aborted transactions are obtainable from the executions of unaborted transactions.

A more problematic question is what process equivalences can be defined. The complication is that the liveness properties of processes may be quite different unless one also considers uncommitted effects. However considering such effects introduces the possibilities of negative effects (undoing message sends due to aborts) and it is not clear how to obtain a fixed point. This remains an interesting topic for future work.

6. ADDING MOBILITY: THE ATF$_\mathrm{M}$ CALCULUS

As noted in Sect. 3.1, the ATF Calculus is first-order, in the sense that processes cannot be transmitted as part of the message payload. Allowing processes in messages is straightforward, but complicates the meta-theory. Calculi such as the π-calculus and the ambient calculus avoid higher-order processes by passing "trigger channels" in messages (in the π-calculus) or moving a process through the network (in the ambient calculus). In this section we consider how a similar approach to passing processes in messages may be incorporated into the ATF calculus. We call this extension of the ATF Calculus the ATF$_M$ Calculus. This extension introduces some redundancy, principally with respect to the encryption operations. The base calculus has the encryption operations of the Spi-calculus [1], while the extension considered here also uses the approach to encryption of the ambient calculus [8]. On the other hand, the encoding of process transmission here, motivated by a desire to keep values and processes separate, is not as natural as the other operations in the base calculus, so we treat process transmission as an extension of this base calculus.

The main addition to the calculus is a message sending operation that includes a process in the payload. There are two operations for accessing such a message: a *forward* operation for forwarding the message to another destination without accessing the message itself, and an *open* operation for "liberating" the process payload from the message. The latter operation is similar to the *open* primitive in the ambient calculus, but extended to avoid any race condition between dissolution and receipt of any communication from that process, and also extended to incorporate atomic failure semantics. The extensions to the syntax are:

$$\begin{array}{rcll}
P \in \text{Process} & ::= & \mathit{send}(M,F) & \text{Message Proc Send} \\
& | & \mathit{forward}(M,M',P_1,P_2) & \text{Message Proc Forward} \\
& | & \mathit{open}(M,M',P_1,P_2) & \text{Message Proc Open} \\
T \in \text{Type} & ::= & T \to () &
\end{array}$$

Although types include process types $T \to ()$, processes are still second-class by a syntactic distinction: continuations F are not values V. Process types are used in typing the operations for forwarding and opening messages containing processes, although process values are not exchanged as part of these operations.

The type rules are best understood by first considering the reduction rules for these operations. First, there are two reduction rules that allow messages containing processes to navigate the network, similarly to the RED IN PLACE and RED OUT PLACE reduction rules in Sect. 4.4. We omit these rules for lack of space; they are reasonably obvious.

The next reduction rule allows a message containing a process to be forwarded to a new destination. This destination is identified by a path in and out of places, and an eventual channel into which the process should be deposited. The forwarding operation requires a capability for reading the channel into which the message has been deposited. It is assumed that the message is encrypted, and the capability constructed by the forwarding operation ensures that the message arrives still encrypted at the new destination:

$$\frac{\begin{array}{c} N^{\text{forw}} \equiv t_1\{\mathit{forward}(\mathit{out}\ n^{\text{ch}}, M, P_1, P_2)\} \\ N^{\text{mesg}} \equiv t_2\{\mathit{send}(\mathit{in}\ n^{\text{ch}}.\mathit{in}\ n^{\text{pk}}, F)\} \\ N' \equiv t_1\{P_1 \mid \mathit{send}(M.\mathit{in}\ n^{\text{pk}}, F)\} \\ \mathcal{L} \not\models t_1\ \textbf{aborted} \quad \mathcal{L} \not\models t_2\ \textbf{committed} \quad \mathcal{L}' \equiv \mathcal{L}; (N^{\text{forw}} \mid N^{\text{mesg}}) \end{array}}{n^{\text{pl}}[\mathcal{L}, (N^{\text{forw}} \mid N^{\text{mesg}} \mid N)] \longrightarrow n^{\text{pl}}[\mathcal{L}', (N' \mid N)]}$$

(RED FORWARD)

Finally we have the rule for "dissolving" a message containing a process. The message is deposited in a channel, so the opening process must have a capability for reading that channel. The message is also assumed to be encrypted, so the opening process must have a capability for decrypting the message. We must also avoid any potential race condition caused by the fact that dissolving the message and communicating with the process so invoked do not constitute an atomic action. This is why the message is a continuation, abstracting over a capability for a communication channel. The opening process provides this capability as part of the opening operation, with the expectation that this is a private channel for initiating communication between the liberated process and the opening process. Finally the liberated process must execute within a transaction, so it executes in the transaction of the process that opened the message:

$$\frac{\begin{array}{c} N^{\text{open}} \equiv t_1\{\mathit{open}(\mathit{out}\ n^{\text{ch}}.\mathit{out}\ n^{\text{pk}}, M, P_1, P_2)\} \\ N^{\text{mesg}} \equiv t_2\{\mathit{send}(\mathit{in}\ n^{\text{ch}}.\mathit{in}\ n^{\text{pk}}, F)\} \\ N' \equiv t_1\{F(M) \mid P_1\} \\ \mathcal{L} \not\models t_1\ \textbf{aborted} \quad \mathcal{L} \not\models t_2\ \textbf{committed} \quad \mathcal{L}' \equiv \mathcal{L}; (N^{\text{open}} \mid N^{\text{mesg}}) \end{array}}{n^{\text{pl}}[\mathcal{L}, (N^{\text{open}} \mid N^{\text{mesg}} \mid N)] \longrightarrow n^{\text{pl}}[\mathcal{L}', (N' \mid N)]}$$

(RED OPEN)

Note that this form of synchronization with a process, liberated from a message payload, does not obviate the need for the value-passing communication channels that are already part of the ATF Calculus. These channels are still necessary for communication between the liberated process and the opening process.

There are also timeout rules for the *forward* and *open* operations:

$$\mathit{forward}(\mathit{out}\ n^{\text{ch}}, M, P_1, P_2) \rightarrow P_2 \quad \text{(RED FORWARD TIMEOUT)}$$

$$\mathit{open}(\mathit{out}\ n^{\text{ch}}.\mathit{out}\ n^{\text{pk}}, M, P_1, P_2) \rightarrow P_2 \quad \text{(RED OPEN TIMEOUT)}$$

With these reduction rules, we are in a position to provide the type rules for the operators. The process-containing message requires that the process in fact be a continuation, parameterized over a capability for a channel for receiving values of type T. The capability associated with the message terminates with a key for encrypting the message. The forwarding message requires a capability for accessing the channel into which the message has been deposited (in encrypted form), as well as a capability for a channel to which the message is to be forwarded. Both of these capabilities have the same type, since the type system does not distinguish input and output capabilities. Finally the open operation requires a capability for accessing a channel and descrypting the message. The message payload should be a continuation parameterized over a capability for a private communication channel, to be supplied by the opening process.

$$\frac{\Gamma \vdash M : \mathit{Cap}[\mathit{Packet}[\mathit{Cap}[\mathit{Chan}[T]] \to ()]] \quad \Gamma \vdash F : \mathit{Cap}[\mathit{Chan}[T]] \to ()}{\Gamma \vdash \mathit{send}(M,F) : ()} \qquad \text{(PROC PSEND)}$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \mathit{Cap}[\mathit{Chan}[\mathit{Packet}[\mathit{Cap}[\mathit{Chan}[T]] \to ()]]] \\ \Gamma \vdash M' : \mathit{Cap}[\mathit{Chan}[\mathit{Packet}[\mathit{Cap}[\mathit{Chan}[T]] \to ()]]] \\ \Gamma \vdash P_1 : () \quad \Gamma \vdash P_2 : ()\end{array}}{\Gamma \vdash \mathit{forward}(M,M',P_1,P_2) : ()} \qquad \text{(PROC FORWARD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \mathit{Cap}[\mathit{Packet}[\mathit{Cap}[\mathit{Chan}[T]] \to ()]] \\ \Gamma \vdash M' : \mathit{Cap}[\mathit{Chan}[T]] \quad \Gamma \vdash P_1 : () \quad \Gamma \vdash P_2 : ()\end{array}}{\Gamma \vdash \mathit{open}(M,M',P_1,P_2) : ()} \qquad \text{(PROC OPEN)}$$

7. RELATED WORK AND CONCLUSIONS

Numerous process algebras have been proposed as the foundations of programming languages for wide-area applications. Most of this work is based on mobile computation and mobile code to deal with latency and firewall problems [12, 8, 10, 26, 17]. Much of this work has focused on access control for mobile computation in networks, as well as tracking the trustworthiness of hosts. Although some work has looked at failures [3, 13, 25, 4], this work has assumed a fail-stop model of failures that is unrealistic in asynchronous distributed systems.

There is a superficial similarity between parts of the ATF-Calculus and the LF-Calculus of Riely and Hennessy [25]. The latter work is an extension of CCS with locations and failures, and it is also a two-tier calculus where processes execute at specific locations. Operations include an operation for "killing" a location, including all processes executing at that location, and operations for testing if a location is live. However these similarities are only superficial, since places play the rôle of locations in our system and transactions are a construct for controlling the granularity of causal dependencies. There is no notion in the LF-Calculus of committing effects or of using logs to undo effects. The ATF Calculus was developed independently of the latter work.

The Argus distributed programming language [20] incorporated atomic remote procedure calls based on the use of nested transactions [21, 24]. The semantics of nested transactions generalize the original database semantics of transactions (for example, correctness is again based on serializability and recoverability [5]). An important difference between nested transactions and the model of transactions considered here is that we allow interference between transactions, whereas the serializability correctness criterion for nested transactions requires that sibling transactions be isolated from each other. Since parent and child transactions share locks, a distinction is made between leaf transactions that access data objects and transactions that spawn other transactions, so a parent transaction never interferes with a descendant transaction. The only causal dependency allowed is the dependency of a child transaction on the parent transaction that created it. We provide a more relaxed correctness criterion, leaving the correctness of concurrency control to the application. Haines et al [15] describe a subroutine library in ML that enhances applications with transactional semantics, with the semantics of nested transactions. Their library provides undoability and persistence as orthogonal features, however they do not give a semantics for these features (while acknowledging that there is interaction between the features).

It is straightforward to add an operation to the ATF Calculus for creating new transactions (with child transactions depending causally on parent transactions). We intend to investigate this further, in the context of considering more flexible ways for creating and combining transactions.

References

[1] Martin Abadi and Andrew Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[2] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The Switchware active network architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):29–36, 1998.

[3] R.M. Amadio and S. Prasad. Localities and failures. In P. S. Thiagarajan, editor, *Proceedings of 14th Conference on Foundations of Software Tech-*

nology and Theoretical Computer Science, number 880 in Lecture Notes in Computer Science, pages 205–216. Springer-Verlag, 1995.

[4] Roberto Amadio. An asynchronous model of locality, failure, and process mobility. In *Coordination '97*, 1997.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[7] Luca Cardelli. Abstractions for mobile computation. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[8] Luca Cardelli and Andrew Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[9] Luca Cardelli and Andrew Gordon. Types for mobile ambients. In *Proceedings of ACM Symposium on Principles of Programming Languages*, San Antonio, January 1999. ACM Press.

[10] Guiseppe Castagna and Jan Vitek. A calculus of secure mobile computations. In *Internet Programming Languages*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[12] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 1996. ACM.

[13] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, August 1996. Springer-Verlag. LNCS 1119.

[14] V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. In B. Simons and A. Z. Spector, editors, *Fault-Tolerant Distributed Computing*, volume 448 of *Lecture Notes in Computer Science*, pages 201–208. Springer-Verlag, 1990.

[15] N. Haines, D. Kindred, J. G. Morrisett, and S. M. Nettles. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, 16(6):1719–1736, November 1994.

[16] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.

[17] Matthew Hennessy and James Riely. Type-safe execution of mobile agents in anonymous networks. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[18] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Proceedings of ACM International Conference on Functional Programming*. ACM Press, September 1998.

[19] Butler Lampson. Atomic transactions. In B. Lampson, M. Paul, and H. Siegert, editors, *Distributed Systems–Architecture and Implementation*, volume 205 of *Lecture Notes in Computer Science*, pages 246–285. Springer-Verlag, 1981.

[20] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3), March 1988.

[21] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan-Kaufman, 1994.

[22] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[23] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Computer and Systems Sciences*, pages 203–246. Springer-Verlag, 1993.

[24] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.

[25] James Riely and Matthew Hennessy. Distributed processes and location failures. In *Proceedings of the International Conference on Automata, Languages and Programming*, 1997.

[26] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1999.

[27] Davide Sangiorgi. Asynchronous process calculi: The first-order and higher-order paradigms. *Theoretical Computer Science*, 1999.