Abstractions for Fault Tolerant Global Computing

Tom Chothia and Dominic Duggan *

Department of Computer Science
Stevens Institute of Technology
Hoboken, NJ 07030
USA.

Abstract

Global computing (WAN programming, Internet programming) distinguishes itself from local computing (LAN computing) by the fact that it exposes some aspects of the network to the application, rather than seeking to hide them with network transparency, as in LAN programming. Global computing languages seek to provide useful abstractions for building applications in such environments. The lqp(·)-calculus is a family of programming languages that use the abstraction of *logs* to specify application-specific protocols for distributed agreement and fault tolerance in global applications. Reflecting the motivation for global computing, the abstraction of logs isolates the communication requirements of such protocols. Two specific instances of the lqp(·)-calculus are provided, the lqp(dc)-calculus and the lqp(dcu)-calculus. These are intended as kernel programming languages for fault-tolerant distributed programming. The calculi incorporate various abstractions for fault tolerance, from which several forms of distributed transactions and optimistic computation may be built. As an example application, a calculus of atomic failures is presented, the atf-calculus, and its encoding in the lqp(dc)-calculus used to verify a correctness property.

1 Introduction

Global computing, sometimes referred to as wide-area computation, wide-area network programming or Internet programming [1], poses interesting challenges for application developers. This is because the traditional programming environments for distributed application development are based on applications spanning local-area networks and enterprise intra-networks. The characteristics of local computing

* Corresponding author.
Email address: {tomc,dduggan}@cs.stevens-tech.edu (Tom Chothia and Dominic Duggan).

environments are different from those of global computing, and suggest a need for different approaches.

Local computing languages are organized around the principle of location transparency, hiding the network behind the abstraction of RPC or RMI [2]. In contrast global computing languages expose the network to the application, recognizing that dealing with the network is an important aspect of global applications, and seeking to provide useful high-level abstractions for application developers. A commercial example of this is given by the Java Jini system [3], while a formal approach is given by the Ambient Calculus [4].

Work on the semantics of global computing languages has focused on mobility of various kinds [4–8]. There has been little attention paid to providing support for fault tolerance, aside from work based on fail-stop failure models, that may not always be appropriate in global computing [1]. An example of a local computing language that provided support for fault tolerance is the Argus language [9]. Fault tolerance was based on guardians and nested transactions [10,11]. Similar support for transactions was provided by languages such as Avalan/C++ and Venari/ML [12,13], and is an integral part of various well-known distributed computing platforms, including CORBA OTS, COM MTS, and Java Jini and JavaBeans [14].

There are two aspects of transactions, as a tool for building fault-tolerant global applications, that we wish to address:

(1) The first aspect is the somewhat monolithic concept of transactions themselves. Transactions include notions of failure atomicity, concurrency control, persistence, and undoing of effects. This particular combination is useful for the kinds of database applications for which transactions were originally designed. It is not clear that this particular combination, or any particular combination, is appropriate for all global applications. For example there are many variants of transactions that have been proposed for various other classes of applications, particularly for long-lived applications [15].

    In this paper we propose a set of largely orthogonal abstractions, that can be combined to build different classes of fault-tolerant applications. Transactions are one of the mechanisms that can be built through such a combination. Fig. 1 provides our set of abstractions.

(2) At the heart of mechanisms for building fault-tolerant applications are some collections of protocols, that in turn rely on a communication system for delivering protocol messages. In global computing, establishing communication channels may itself be an important part of a global application. For example Internet communication must nowadays navigate through firewalls, proxies, network address translators and load balancers. Currently this is done in an *ad hoc* fashion, in a way that violates data abstraction and the supposed layering of protocols. The ambient calculus is based on the notion of applications explicitly navigating administrative domains delimited by firewalls. If

fault-tolerance protocols are part of the underlying support for building global applications, how are the protocol messages to be delivered in a semantically correct (and secure) manner?

In this paper we make some progress towards providing an answer to this issue, by isolating the communication requirements of protocols in the semantics of the underlying abstractions for building fault-tolerant global applications. There is a notion of stable storage and logs; processes may query and extend their own local logs, but may only query the logs of remote sites. The issue of how to query the logs of a remote site, without relying on an underlying communication infrastructure, is left to a sequel.

The Jini system's [3] support for fault tolerance provides interfaces for the two-phase commit protocol [16], and a default implementation of the protocol. We take the position that this approach is too high level and too protocol-specific. It is too high level in the sense that atomic commitment is in general impossible to achieve in asynchronous distributed systems [17,18]. On the other hand, building a particular (potentially blocking) protocol for atomic commitment into the language semantics overcommits the language design to particular implementations. Our emphasis is instead on isolating the invariants that should be maintained by stable storage, and providing mechanisms that applications can use to change stable storage in such a way that these invariants are preserved. Protocols such as two-phase commit can then be provided as libraries on top of these primitives. As such, our approach is akin to a type system for a global programming environment, albeit one where the invariant-preserving operations on stable storage are checked at run-time. This is represented in our calculus by operations for appending to logs, that require preconditions to be satisfied before such an appending is allowed.

At the core of the abstractions in Fig. 1 is the concept of *process groups*. We refer to such a process group as a *conclave*. A conclave is at its simplest level a group of processes

$$c\{P_1 \mid \dots \mid P_k\}$$

where the conclave identifier c serves as a name for the group. The processes in a conclave share a *log*, which only they may modify. Logs are used to specify protocols for global agreement, on an application-specific basis. The operations for adding log entries are restricted to ensure certain global invariants are preserved; what those invariants are depends on the particular protocols being specified. In fact we obtain a family of calculi, all sharing the same kernel language, the lqp($\cdot$)-calculus. Each instance in this family is obtained by adding a particular set of log entry types, and associated rules for appending new log entries, to the lqp($\cdot$)-calculus.

Conclaves can also be considered as a unit of *atomic failure*. When we consider using conclaves to model transactions, the intention is that if any process in a conclave "fails," then all of the processes in that conclave "fail." We identify *causality* as the fundamental yardstick for measuring dependencies between the failures of

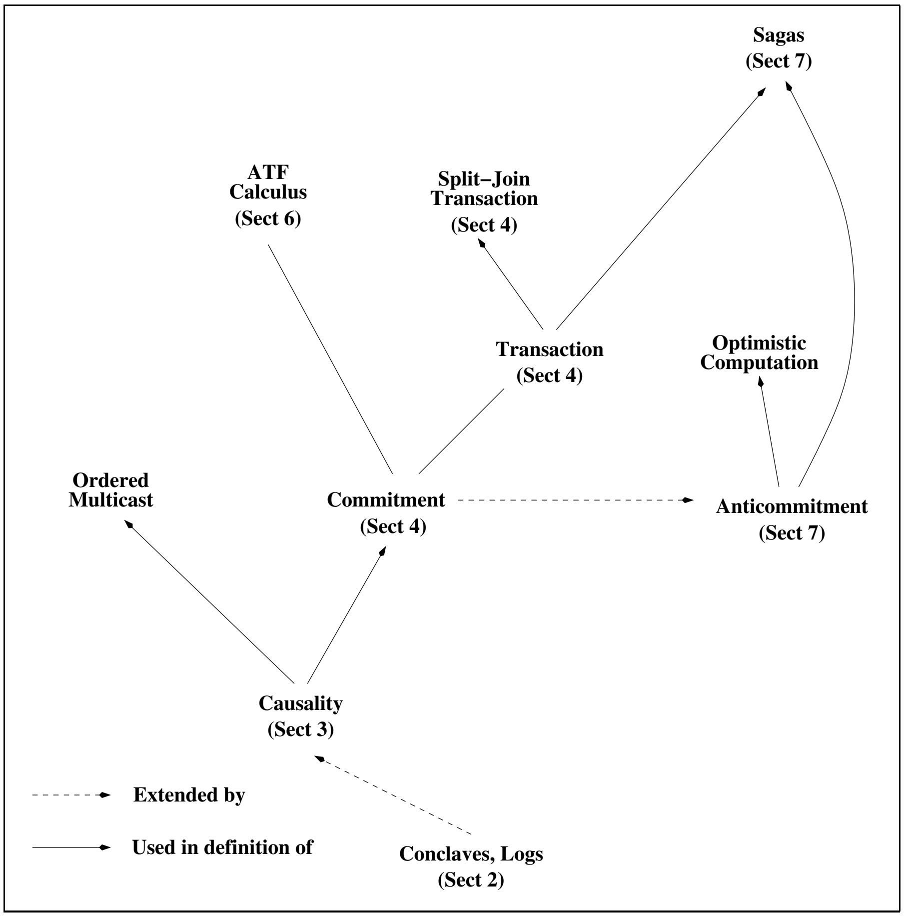


Fig. 1. Applications of Abstractions

different conclaves, and we propose *causal consistency* as a correctness criterion for proper executions involving atomic failures. If a conclave c_1 consumes some of the output of another conclave c_2, this establishes a causal dependency from c_2 to c_1. If c_2 subsequently aborts, then c_1 must also abort. The intuition of causal consistency is that, as with traditional transactions, a run of conclaves with failures should be in some sense equivalent to one in which no failures occurred, or in which at least no effects besides failure are visible for the failed conclaves (the latter alternative is possible with nested transactions). By "visible" we mean informally that no database updates or messages issued by failed processes are observed by processes outside the corresponding conclaves. Rather than restricting causal relationships to tree structures, we allow arbitrary directed graph structures, including cyclic graphs.

In Sect. 2 we review the basic mechanisms in the lqp(·)-calculus, including the message-passing primitives of the pi-calculus, and the notion of process groups and logs added by the lqp(·)-calculus. In Sect. 3–Sect. 4 we provide the lqp(dc)-calculus, an instance of the lqp(·)-calculus that provides dependencies and commitment. We use this to model transactional mechanisms for distributed programming. We verify a notion of correctness for the lqp(dc)-calculus in Sect. 5. As an extended example, we provide a calculus of atomic failures, the atf-calculus, in Sect. 6. We provide an encoding of the atf-calculus in the lqp(dc)-calculus, and use the correctness of the latter to verify a correctness property for the former. In Sect. 7 we describe the lqp(dcu)-calculus, an extension of the lqp(dc)-calculus with mechanisms for atomic *anti*commitment, undoing the effects of conclaves that have been committed optimistically. In Sect. 8 we verify the correctness of the lqp(dcu)-calculus. Finally Sect. 9 provides a comparison with related work and conclusions.

We deliberately do not consider any particular failure model in our work [19–23], since they are orthogonal to the issues addressed in this paper. Although there are well-known impossibility results for achieving agreement in asynchronous distributed systems [17,18], these results can be circumvented by making further assumptions about the environment (e.g., partial synchrony, unreliable failure detectors [19]) or by weakening the correctness requirements of the protocols used to achieve distributed agreement. Indeed the whole point of this work is that the application developer can use the approach presented here with any failure model and any protocol correctness conditions that she considers appropriate to the particular application. For example, a traditional transaction system may rely on unreliable failure detectors, aborting an uncommitted transaction if it times out while waiting to be contacted for a run of the commitment protocol. Other variants may rely on a fail-stop model to detect a crashed protocol administrator and elect a new administrator (for example, using three-phase commit); the protocol then fails to work when the network is partitioned. All of these approaches, and others based on more powerful unreliable failure detectors, can be added to our model. But we take the point of view that this should be done on an application-by-application basis, rather than building particular failure models or consistency conditions into the language.

2 Logs and Conclaves

This paper does not introduce one calculus, but a family of calculi. Each calculus in this family shares certain features:

(1) An abstraction of *logs* that are used to specify protocols for global agreement. Each calculus in the family specifies a particular set of log entry types, correctness conditions for each form of log entry, and a collection of log append rules that should be verified to preserve correctness.
(2) A notion of *conclaves*, process groups that share the same logs. All processes execute within a conclave. There is an implicit notion of locality associated with conclaves; all processes executing in a conclave should be on the same machine since they may all change the same shared log.

The common core of the family of calculi is described by the lqp($\cdot$)-calculus. This calculus has no log entry types, only the common structure used by all calculi in the

$$
\begin{array}{rcll}
v \in \text{Value} & ::= & x & \text{Variable} \\
& | & n, c & \text{Name} \\
P \in \text{Process} & ::= & \mathsf{stop} & \text{Stopped process} \\
& | & \mathsf{send}\ v!\langle v_1, \ldots, v_k \rangle & \text{Message} \\
& | & \mathsf{receive}\ n?(x_1, \ldots, x_k);\ P & \text{Message receive} \\
& | & \mathsf{new}\ n;\ P & \text{Scoped name} \\
& | & \mathsf{repeat}\ P & \text{Replication} \\
& | & (P_1 \mid P_2) & \text{Parallel composition}
\end{array}
$$

(a) Core pi-calculus

$$
\begin{array}{rcll}
P \in \text{Process} & ::= & \mathsf{logif}\ c\{\!\{L\}\!\}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 & \text{Check for log entry} \\
& | & \mathsf{logawait}\ (x_1, \ldots, x_n) c\{\!\{L\}\!\};\ P & \text{Wait for log entry} \\
& | & \mathsf{logappend}\ \langle \overline{v_k} \rangle\ \mathsf{with}\ \textit{rule-name};\ P & \text{Append to log} \\
& | & c\{P\} & \text{Fork new conclave} \\
& | & \mathsf{loginit};\ P & \text{Initialize the log} \\
C \in \text{Net} & ::= & c\{P\} & \text{Process in conclave} \\
& | & c\{\!\{L\}\!\} & \text{Log} \\
& | & \mathsf{new}\ n;\ C & \text{Scoped name} \\
& | & (C_1 \mid C_2) & \text{Parallel composition, wire} \\
L \in \text{Log} & ::= & \mathsf{true} \ \mid\ L_1 \wedge L_2 & \text{Empty log, conjunction}
\end{array}
$$

(b) Extensions for the lqp($\cdot$)-calculus

Fig. 2. Syntax of the lqp($\cdot$)-calculus

family. The syntax for the lqp($\cdot$)-calculus is given in Fig. 2. As usual in such calculi, processes are simple "assembly language" concurrent programs, with operations for message-passing. In this case the language is an extension of the asynchronous pi-calculus [24–26], a popular calculus for describing distributed programs where channel names are globally unique and may be transmitted in messages. For example a client-server application can be described by having the client send a private reply channel to the server. In addition to constructs for sending and receiving messages, there is also an operation for generating new channel names, for replicating processes (this can be used to define recursive process descriptions) and for forming the parallel composition of processes. For formal reasoning purposes, messages are restricted to tuples of values.

$$c\{\mathsf{stop}\} \mid C \equiv C \qquad \mathsf{stop} \mid P \equiv P$$

$$C_1 \mid C_2 \equiv C_2 \mid C_1 \qquad P_1 \mid P_2 \equiv P_2 \mid P_1$$

$$(C_1 \mid C_2) \mid C_3 \equiv C_1 \mid (C_2 \mid C_3) \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$$

$$\mathsf{new}\ n_1;\ \mathsf{new}\ n_2;\ C \equiv \mathsf{new}\ n_2;\ \mathsf{new}\ n_1;\ C \qquad \mathsf{new}\ n_1;\ \mathsf{new}\ n_2;\ P \equiv \mathsf{new}\ n_2;\ \mathsf{new}\ n_1;\ P$$

$$\mathsf{new}\ n;\ C \equiv C,\ \ n \notin fn(C) \qquad \mathsf{new}\ n;\ P \equiv P,\ \ n \notin fn(P)$$

$$c\{\mathsf{new}\ n;\ P\} \equiv \mathsf{new}\ n;\ c\{P\},\ n \neq c \qquad c\{P_1 \mid P_2\} \equiv c\{P_1\} \mid c\{P_2\}$$

$$\mathsf{true} \wedge L \equiv L \quad L_1 \wedge L_2 \equiv L_2 \wedge L_1 \quad (L_1 \wedge L_2) \wedge L_3 \equiv L_1 \wedge (L_2 \wedge L_3)$$

$$(\mathsf{new}\ n;\ C_1) \mid C_2 \equiv \mathsf{new}\ n;\ (C_1 \mid C_2),\ \ n \notin fn(C_2)$$

$$(\mathsf{new}\ n;\ P_1) \mid P_2 \equiv \mathsf{new}\ n;\ (P_1 \mid P_2),\ \ n \notin fn(P_2)$$

$$\mathsf{repeat}\ P \equiv P \mid \mathsf{repeat}\ P$$

Fig. 3. Equivalence Rules for the lqp($\cdot$)-calculus

In the version of the asynchronous pi-calculus used here and elsewhere [26], processes can communicate output capability, but not input capability. The join calculus has a similar restriction [5]. There are both theoretical and practical reasons for this restriction [26, Sect. 2.5.2]. For our purposes, the reason for making this restriction is that otherwise an implementation of our calculus would require atomic commitment (presumably using two-phase commit), which would of course defeat the whole point of the calculus. As long as an input capability is not shared between different network sites, the restriction ensures that this invariant is preserved during execution. There are sound pragmatic reasons for weakening this restriction to allow transmission of input capability within a conclave, since conclaves imply locality [27]; however we eschew pursuing this elaboration in this presentation.

A process P executing as part of a conclave c is represented by the network term $c\{P\}$. These network terms represent a group of processes that share a log. So each conclave should run on a single host, to avoid the need for distributed coordination as a primitive in the language. Defining protocols for distributed coordination is after all the whole point of defining the lqp($\cdot$)-calculus. A network C is a collection of processes executing in conclaves, and the logs associated with the conclaves.

Besides organizing processes into conclaves, the other innovation in this calculus is the addition of logs. A log in the lqp($\cdot$)-calculus is represented by a multiset of located propositions, abstractly representing the entries in the "log." Each conclave has a log, represented by a collection of logical propositions L. The fact that a log is for a conclave named c is represented by a "located" log of the form $c\{\!\{L\}\!\}$. We require consistent networks to have exactly one log for each conclave, as shown in Sect 5. Some of the log predicates are *derived predicates*, meaning that there will in general not be log entries of those forms (except for caching purposes). Rather they represent queries that may be made of a remote site.

$$(c_1\{\mathsf{send}\ n!\langle\overline{n_k}\rangle\} \mid c_2\{\mathsf{receive}\ n?(\overline{x_k});\ P\}) \longrightarrow c_2\{\{\overline{n_k}/\overline{x_k}\}P\} \qquad \text{(RED RECEIVE)}$$

$$\frac{c\{\!\{L'\}\!\} \models c\{\!\{\{\overline{n_k}/\overline{x_k}\}L\}\!\}}{(c\{\!\{L'\}\!\} \mid c'\{\mathsf{logawait}\ (\overline{x_k})c\{\!\{L\}\!\};\ P\}) \longrightarrow (c\{\!\{L'\}\!\} \mid c'\{\{\overline{n_k}/\overline{x_k}\}P\})} \qquad \text{(RED WAIT)}$$

$$\frac{c\{\!\{L'\}\!\} \models c\{\!\{L\}\!\}}{(c\{\!\{L'\}\!\} \mid c\{\mathsf{logif}\ c\{\!\{L\}\!\}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\}) \longrightarrow (c\{\!\{L'\}\!\} \mid c\{P_1\})} \qquad \text{(RED IFLOGTRUE)}$$

$$\frac{c\{\!\{L'\}\!\} \not\models c\{\!\{L\}\!\}}{(c\{\!\{L'\}\!\} \mid c\{\mathsf{logif}\ c\{\!\{L\}\!\}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\}) \longrightarrow (c\{\!\{L'\}\!\} \mid c\{P_2\})} \qquad \text{(RED IFLOGFALSE)}$$

$$\frac{\begin{array}{c} C = (C' \mid c\{\mathsf{logappend}\ \langle\overline{v_k}\rangle\ \mathsf{with}\ \textit{rule-name};\ P\} \mid c\{\!\{L_1\}\!\}) \\ C, c \models (\overline{v_k}) \xrightarrow{\textit{rule-name}} \mathsf{new}\ \overline{n_m};\ L_2 \qquad \overline{n_m} \cap \mathit{fn}(L_1) = \{\} \end{array}}{C \longrightarrow (C' \mid c\{P\} \mid \mathsf{new}\ \overline{n_m};\ c\{\!\{L_1 \wedge L_2\}\!\})} \qquad \text{(RED APPEND)}$$

$$\frac{C = \prod \overline{c_k\{P_k\}} \qquad c \notin \{\overline{c_k}\}}{\mathsf{new}\ c;\ (c\{\mathsf{loginit};\ P\} \mid C) \longrightarrow \mathsf{new}\ c;\ (c\{\!\{\mathsf{true}\}\!\} \mid c\{P\} \mid C)} \qquad \text{(RED LOGINIT)}$$

$$\mathbb{E}[\,\cdot\,] ::= [\,\cdot\,] \ \mid\ (\mathbb{E}[\,\cdot\,] \mid C) \ \mid\ \mathsf{new}\ n;\ \mathbb{E}[\,\cdot\,]$$

$$\frac{C_1 \equiv \mathbb{E}[C_1'] \quad C_1' \longrightarrow C_2' \quad C_2 \equiv \mathbb{E}[C_2']}{C_1 \longrightarrow C_2} \ \text{(RED CONG)} \qquad c_1\{c_2\{P\}\} \longrightarrow c_2\{P\} \ \text{(RED EXEC)}$$

(a) Computation Rules

$$\frac{C_i \models c\{\!\{L\}\!\} \text{ for some } i \in \{1,2\}}{(C_1 \mid C_2) \models c\{\!\{L\}\!\}} \ \text{(PRED PAR)} \qquad \frac{n \notin \mathit{fn}(L) \cup \{c\} \quad C \models c\{\!\{L\}\!\}}{(\mathsf{new}\ n;\ C) \models c\{\!\{L\}\!\}} \ \text{(PRED NEW)} \qquad \frac{L \equiv L' \wedge L''}{c\{\!\{L\}\!\} \models c\{\!\{L'\}\!\}} \ \text{(PRED LOG)}$$

(b) Log Query

Fig. 4. Semantics of lqp(·)-calculus (without log append rules)

The semantics for any calculus in the lqp(·)-calculus family are specified using various judgement forms:

$P_1 \equiv P_2\ ,\ L_1 \equiv L_2\ ,\ C_1 \equiv C_2$	Equivalence	Fig. 3
$C_1 \longrightarrow C_2$	Computation	Fig. 4
$C \models c\{\!\{L\}\!\}$	Log query	Fig. 4 and 6
$C, c \models (\overline{x_k}) \xrightarrow{\textit{rule-name}} L$	Log append rule	Fig. 6, 7, 18 and 19

The reflexive transitive closure of the computation relation is denoted by $C_1 \xrightarrow{*} C_2$.

The structural equivalence rules for processes and conclaves are provided in Fig. 3. The rules for processes P are the usual equivalence rules for the pi-calculus, including extrusion of scope of locally generated names. The rules for conclaves replicate the rules for processes, and also include the following:

$$c\{P_1 \mid P_2\} \equiv c\{P_1\} \mid c\{P_2\} \qquad c\{\mathsf{new}\ n;\ P\} \equiv \mathsf{new}\ n;\ c\{P\},\ n \neq c$$

These rules relate processes to conclaves; a collection of processes located in conclaves is equivalent to a collection of "atomic" processes, each executing a local operation (message send or message receive, or an operation on the logs as specified below) located in conclaves. Each such atomic process has the form $c\{P\}$ where P has one of the aforesaid forms and where c denotes the "location" of the process. In this calculus, we take processes up to α-conversion (renaming) of scoped names, so that all substitutions over processes and conclaves are assumed to be capture-avoiding. The forking construct $c\{P\}$ still does not imply any nesting of conclaves, rather it is simply a way for a conclave to extrude code that executes in another conclave. The loginit construct allows an initial empty log to be created for a conclave. The syntactic restriction on the context C, and the static scoping rules for the pi-calculus, ensure that there are no other logs for this conclave present. We use these operations to define a construct in Sect. 4 that allows a new transaction to be defined.

Some operations require examining all of the log entries for a conclave, for example to ensure the absence of a particular log entry. Therefore each conclave c is required to have a single log, of the form $c\{\!\{L\}\!\}$ where L is a conjunction of log entries. There are three constructs for interacting with stable storage:

$$\begin{array}{rl} P ::= & \mathsf{logif}\ c\{\!\{L\}\!\}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \\ \mid & \mathsf{logawait}\ (x_1,\ldots,x_k)c\{\!\{L\}\!\};\ P \\ \mid & \mathsf{logappend}\ \langle v_1,\ldots,v_k\rangle\ \mathsf{with}\ \textit{rule-name};\ P \end{array}$$

The semantics for these constructs, as well as the semantics for message-passing, are provided in Fig. 4(a). The logawait construct blocks until log entries matching the pattern are in stable storage. For example an undo action can be specified to have the form

$$\mathsf{logawait}\ ()c\{\!\{\mathit{Aborted}\}\!\};\ P$$

If the conclave c is aborted, then the process P will execute, presumably undoing some of the effects of the aborted conclave. The transition rule in the operational semantics for the await construct has the form:

$$\frac{c\{\!\{L'\}\!\} \models c\{\!\{\{\overline{n_k}/\overline{x_k}\}L\}\!\}}{(c\{\!\{L'\}\!\} \mid c'\{\mathsf{logawait}\ (\overline{x_k})c\{\!\{L\}\!\};\ P\}) \longrightarrow (c\{\!\{L'\}\!\} \mid c'\{\{\overline{n_k}/\overline{x_k}\}P\})} \quad \text{(RED WAIT)}$$

This uses the general judgement form $C \models c\{\!\{L\}\!\}$ (in this particular case as $c\{\!\{L'\}\!\} \models c\{\!\{\{\overline{n_k}/\overline{x_k}\}L\}\!\}$) to query if the proposition L is present in the logs for the conclave c. The rules for querying the logs are provided in Fig. 4(b). The rules fairly straightforwardly decompose a surrounding context for a log entry of the required form. The general form of the log query rules is useful for checking the preconditions of the log append rules, provided in subsequent sections. In Sect. 7 we allow transactions to anti-commit. This raises the possibility of having two occurrences of the same predicate in a log, with different arguments. Hence the logawait action may be non-deterministic. To avoid any problems with substituting the names from the log, we require that each occurrence of a predicate always has the same number of arguments.

The $\mathsf{logappend}$ construct is used to add to the contents of the log. The operations for adding to the log are specified by named rules. Each rule requires some preconditions and adds some new collection of propositions to the log. The rules are predefined as part of the calculus, in order to ensure some consistency properties of the operational semantics. These log append rules are specified in Fig. 6, 718 and 18, using judgements of the form

$$C, c \models (\overline{v_k}) \xrightarrow{\textit{rule-name}} \mathsf{new}\ \overline{n_m};\ L$$

where *rule-name* is the name of the rule, C the surrounding context (used for checking preconditions), c the name of the conclave executing the rule, L the propositions added to storage by the rewrite rule, and $v_1, \ldots, v_k$ are values (names or variables) that are inputs to the append rule. The names $n_1, \ldots, n_m$ are new names generated as part of the addition of the log propositions; they are the free names in L output by the append rule. This new name generation as part of appending a log entry is used in the lqp(dcu)-calculus, in the undoing of anticommitment in Sect. 7. Then the transition rule for changing storage is given by:

$$\frac{\begin{array}{c} C = (C' \mid c\{\mathsf{logappend}\ \langle\overline{v_k}\rangle\ \mathsf{with}\ \textit{rule-name};\ P\} \mid c\{\!\{L_1\}\!\}) \\ C, c \models (\overline{v_k}) \xrightarrow{\textit{rule-name}} \mathsf{new}\ \overline{n_m};\ L_2 \end{array}}{C \longrightarrow (C' \mid c\{P\} \mid \mathsf{new}\ \overline{n_m};\ c\{\!\{L_1 \wedge L_2\}\!\})} \quad \text{(RED APPEND)}$$

A conclave c can only add to its own log entries. It is possible to check for preconditions in the surrounding context C. This context includes the log for c, $c\{\!\{L_1\}\!\}$. For some rules it may be necessary for a conclave to examine the logs of other conclaves, though only for positive conditions (the presence, but not the absence, of log entries at other sites). In some cases this may require communication with remote sites holding those logs. These remote sites are captured by the part of the context C'. The semantics abstracts from how communication with remote logs should be done. An obvious approach is to send and receive system messages "under the hood," possibly piggybacked on application messages. This assumes the availability of point-to-point communication channels between processes, which may not always be available. We comment on an alternative approach in the conclusions. The point is that the only assumption about the remote communication of state made, in this model, is that processes are able to query the logs of remote conclaves.

We disallow querying for the absence of remote log entries because of obvious race conditions in any implementation of such querying. An implementation that avoided such race conditions would require distributed agreement, which is provably impossible in asynchronous distributed systems [17,18]. For example, we might imagine a protocol where two remote sites compete for a token: a conclave adds a log entry declaring that it has the token if the other conclave has no such log entry. The remote query for the absence of the log entry must be done atomically with the local addition of the log entry. But the implementation of such an atomic operation would require a protocol for distributed mutual exclusion. This is related to the impossibility result of Palamidessi [28], who shows that a distributed election protocol cannot be implemented in the asynchronous pi-calculus. The logif construct allows a process to check for the presence of a particular log entry. There is an else part to check if the log entry is absent. Since we disallow checking for the absence of entries in remote logs, we restrict the logif to only check for the presence or absence of *local* log entries. The logif is used in the encoding of the atf-calculus into the lqp(dc)-calculus, provided in Sect. 6.

Stable storage in our calculus is used to safely save the state of the conclaves, where state is recorded by several proposition types, for causal relationships, commitment protocol state, etc. As a preliminary example, we have specified what it means for a conclave to commit or abort. In a database application, abortion means that updates must be undone and locks released, while commitment means that scheduled updates must be written to the database. We leave the exact semantics of abortion or commitment to the application, and only require that a record be kept in storage of the aborted or committed state of the conclave.

For a conclave that has no causal dependencies with other conclaves, we could supply two rewrite rules, for committing and aborting respectively:

$$\frac{C \not\models c\{\!\{Aborted\}\!\}}{C, c \models () \xrightarrow{\text{CommitEx1}} Committed} \text{(RED COMMIT EX1)} \qquad \frac{C \not\models c\{\!\{Committed\}\!\}}{C, c \models () \xrightarrow{\text{AbortEx1}} Aborted} \text{(RED ABORT EX1)}$$

These rules are only illustrative examples. The actual, more complex log append rules are provided in the following sections.

$L \in \text{Log}$	$::=$	$c_1 \to c_2$	c_1 immediately precedes c_2
	$\mid$	$\mathit{PreClosed} \mid \mathit{Closed}(S)$	No further causal preds
	$\mid$	$\mathit{PreCommitted}$	Commitment protocol
	$\mid$	$\mathit{Committed} \mid \mathit{Aborted}$	Committed, aborted
	$\mid$	$\mathit{IPreds}(S)$	Derived predicates
$S \in \text{Set}$	$::=$	$\{v_1, \ldots, v_k\}$	

Fig. 5. Log Entry Types for the lqp(dc)-calculus

However, these rules are an example of the facility of being able to view all log entries for the local conclave when checking the precondition for the rewrite rules, and therefore being able to check the *absence* of certain log propositions. This ability to view all of the local log entries relies on the invariant of every conclave having a single log.

For a conclave that has causal predecessors, if all of the causal predecessors of that conclave are committed, then the following rewrite rule allows that conclave to itself commit. The antecedent for the following rule checks that any causal predecessors c' of the conclave c have committed:

$$\frac{C \not\models c\{\!\{\mathit{Aborted}\}\!\} \qquad (C \models c\{\!\{c' \to c\}\!\} \text{ implies } C \models c'\{\!\{\mathit{Committed}\}\!\}) \text{ for all } c' \in \mathit{fn}(C)}{C, c \models () \xrightarrow{\text{CommitEx2}} \mathit{Committed}} \quad (\text{RED COMMIT EX2})$$

Here we understand the implication in terms of the classical encoding:

$$(C \models c\{\!\{L\}\!\} \text{ implies } C \models c\{\!\{L'\}\!\}) \equiv (C \not\models c\{\!\{L\}\!\} \text{ or } C \models c\{\!\{L'\}\!\}).$$

So the antecedent for such an implication must only involve local log entries, since our calculus does not allow querying for the absence of remote log entries.

3 Causality

In this section and the next section, we present one particular instance of the lqp($\cdot$)-calculus, the lqp(dc)-calculus. This calculus adds log entry types and log append rules for tracking dependencies between conclaves, and for committing or aborting a collection of conclaves in a way that is consistent with the dependencies between those conclaves. We consider dependencies in this section, and we consider

$$\frac{C \not\models c\{\!\{PreClosed\}\!\}}{C, c \models (c') \xrightarrow{CausalPred} (c' \to c)}$$
(RED CAUSAL PRED)

$$C, c \models () \xrightarrow{PreClosed} PreClosed$$
(RED PRECLOSED)

$$\frac{S = \bigcap\{S' \mid c \in S' \text{ and } \forall c' \in S'.\exists S''.\ (C \models c'\{\!\{IPreds(S'')\}\!\} \text{ and } S'' \subseteq S')\}}{C, c \models () \xrightarrow{Closed} Closed(S)}$$
(RED CLOSED)

(a) Log Append Rules

$$\frac{C \models c\{\!\{c' \to c\}\!\}}{C \models c' \Rightarrow c}$$
(PRED CAUSAL LOG)

$$C \models c \Rightarrow c$$
(PRED CAUSAL REFL)

$$\frac{C \models c_1 \Rightarrow c_2 \quad C \models c_2 \Rightarrow c_3}{C \models c_1 \Rightarrow c_3}$$
(PRED CAUSAL TRANS)

$$\frac{C \models c\{\!\{PreClosed\}\!\} \quad S = \{c' \mid C \models c\{\!\{c' \to c\}\!\}\}}{C \models c\{\!\{IPreds(S)\}\!\}}$$
(PRED PREDS)

(b) Log Query Rules

Fig. 6. Semantics of Causality in the lqp(dc)-calculus

commitment in the next section. The log entry types for the lqp(dc)-calculus are provided in Fig. 5. There is one derived log entry type:

> $c\{\!\{IPreds(S)\}\!\}$ denotes that S is the set of the immediate predecessors of c. This is useful for some of the consistency rules that are used to reason about the correctness of the calculus.

Causality is already recognized in the distributed systems community as important for reasoning about distributed computations, in characterizing global states, computing distributed snapshots, designing fault-tolerant replicated systems etc [29,30]. Traditionally, causality is characterized by dependencies induced by messages exchanged between concurrently executing sequential processes (for example, Lamport's "happened-before" relation [31]). However the approach of tracking causal dependencies at the communication level has been criticized [32,33], both for missing dependencies and for detecting "false" dependencies. The former may happen because of hidden channels outside the communication system (for example, physical pressure in a pipe), while the latter may happen because there is no causal dependency (at the application level) between a message that is sent and a message that was received just before the message send. Cheriton and Skeen [33] argue that what is required is a mechanism for tracking causal dependencies at the application level rather than the communication level, since the application can be aware of hidden channels and can avoid false dependencies.

We use conclaves as a mechanism for tracking causality at the application level. Causality is not a relationship between message send and receive events, but rather is a failure dependency relationship between conclaves: if a conclave fails, then conclaves that depend on it are also required to fail. Furthermore, it would be a mistake to track causal dependencies based on communication between conclaves. For example two conclaves on different machines may communicate via firewall daemons, but we would not expect a firewall daemon to fail because a process whose message it delivered failed. Instead we allow the application itself to assert causal dependencies between conclaves. Because of this, there is no scalable way to prevent cycles in the causal dependency graph.

This has implications for completing distributed conclaves. To maintain causal consistency, a conclave cannot commit until all of its causal predecessors have committed or are also willing to commit. Because of causal cycles, it may be necessary for several conclaves (all of the members of a strongly connected component in the causality graph) to commit simultaneously. We require each conclave to execute at a specific network site, but conclaves may communicate with other conclaves at other sites. So it may be necessary to run an atomic commitment protocol involving several conclaves over an unreliable network. Since this is in general an unsolvable problem, we adopt an approach that can be the basis for widely-used protocols such as two-phase commit and early-prepare commit, but it is not tied to any particular implementation.

When a transaction aborts, its effects must be undone. If changes have been made to a database, the previous values of the changed variables must be restored. We do not provide automatic support for undoing the effects of conclaves that abort. Except for the special case of databases, it is not clear what form undoing should take. For example, undoing a transfer of funds may involve sending an attorney's letter through the ordinary mail. In any case if the receiver of the original message has accepted a causal dependency on the sender of the message, the receiver will be prevented from completing.

Causality is recorded in storage using located propositions of the form $c\{\{c' \rightarrow c\}\}$, recording (in c's logs) that c' is a causal predecessor of c. These propositions are added by processes using the logappend construct, using the (RED CAUSAL PRED) transition rule in Fig. 6. The causal log entries give rise to a reflexive transitive relation $c_1 \Rightarrow c_2$, with rules given in Fig. 6.

Some decisions must be made based on the assumption that all causal predecessors of a conclave are known, i.e., causal predecessors cannot be added after such decisions are made. Therefore we add another log entry type $c\{\!\{Closed(S)\}\!\}$ that denotes that no further causal predecessors can be added for the conclave c, and that S (a set of conclave names) is the set of all causal predecessors of c. We also add another proposition $Closed(\_)$ that is derived from the former proposition type, and simply denotes that the set of causal predecessors is closed.

A conclave cannot autonomously close up the set of its causal predecessors, because one of its predecessors may be open to causal extensions. A conclave could wait until each of its predecessors are closed before it closes itself up to further extensions. However this may lead to deadlock if there are causal cycles. We therefore add a buffer state, *PreClosed*, that a conclave can transition to unconditionally. A transition to the *Closed* state is enabled when all of the causal predecessors of a conclave, including the conclave itself, are in the *PreClosed* state. This is given by the (RED CLOSED) transition in Fig. 6.

In the latter transition, the set S is the set of all causal predecessors, transitively closed, of the conclave c:

(1) The root c is required to be in S.
(2) For any $c' \in S$, the log of c' should be available in the context C, and c' should be preclosed. Let S'' be the set of its immediate predecessors. Then all of these should be in the transitive closure S.
(3) No conclave name should be in S unless it is required to be by one of the two rules above.

The transition rule uses the derived predicate $IPreds(S_i)$ to check that the conclave c_i is "preclosed" and that the current (and therefore permanent) set of c_i's immediate predecessors is S_i. These two checks must be done both to ensure that the set of immediate predecessors of c_i is not enlarged between the time that it is reported to c, and the time that c_i becomes preclosed. The set of predecessors of c is then the smallest set containing c and closed under the addition of these sets of immediate predecessors.

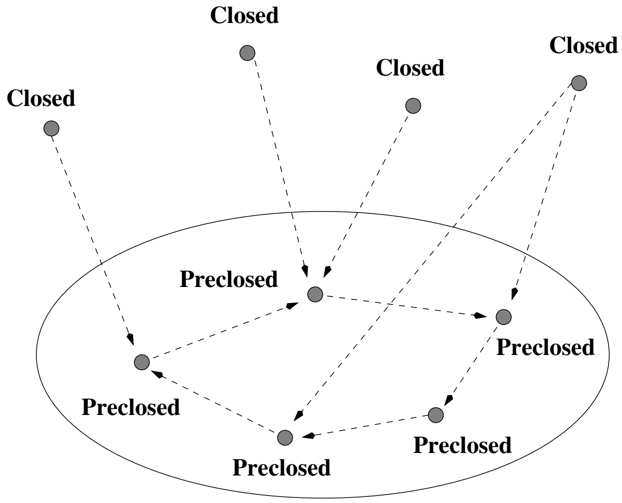
Other protocols for causal closure can be built on top of this framework, using the transitivity property of causality to trim the logs. For example, a conclave must have evidence that all of its causal predecessors, not just its immediate causal predecessors, are either in the closed or preclosed state. Using the property that if a conclave is causally closed, then all of its causal predecessors are closed, a protocol can restrict its attention to all of the conclaves in a strongly connected component of the causality graph. A coordinator for the strongly connected component can use a two-phase commit protocol to check that all conclaves in the component are preclosed, and that all immediate causal predecessors outside the component are closed, to authorize the transition of the conclaves in the component to the closed state:

$$\frac{C \not\models c\{\!\{PreCommitted\}\!\}}{C, c \models () \xrightarrow{AtStAbort} Aborted} \text{(RED AT STABORT)} \qquad \frac{C \models c' \Rightarrow c \quad C \models c'\{\!\{Aborted\}\!\}}{C, c \models () \xrightarrow{AtPcAbort} Aborted} \text{(RED AT PCABORT)}$$

$$\frac{C \models c\{\!\{Closed(\_)\}\!\} \quad C \not\models c\{\!\{Aborted\}\!\}}{C, c \models () \xrightarrow{AtStPreCommit} PreCommitted} \quad \text{(RED AT STPRECMT)}$$

$$\frac{C \models c\{\!\{Closed(S)\}\!\} \quad \forall c' \in S.\ C \models c'\{\!\{PreCommitted\}\!\}}{C, c \models () \xrightarrow{AtPcCommit} Committed} \quad \text{(RED AT PCCOMMIT)}$$

Fig. 7. Semantics of Commitment in the lqp(dc)-calculus

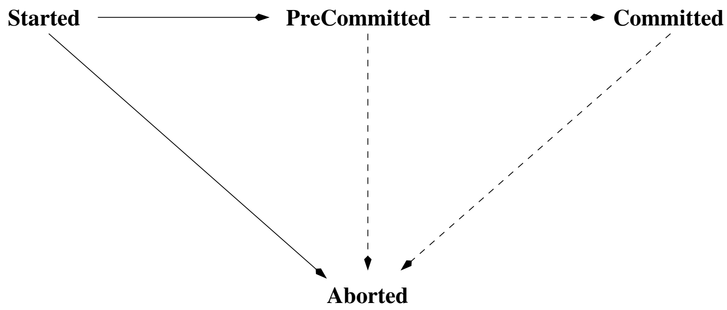


In this picture, nodes represent conclaves in various states (causally closed, prepared to close), edges represent causal dependencies, and the large oval represents a collection of mutually dependent conclaves that can cooperate using some atomic commitment protocol to achieve causal closure, using the fact that predecessor conclaves outside this collection are already closed.

4 Commitment

A conclave encapsulates a set of processes that perform some set of actions that eventually either succeed or fail. We refer to these alternatives as *commitment* and *abortion*. Commitment builds on causality, since a conclave cannot commit unless all of its causal predecessors commit. It is tempting to define a log append rule that allows a conclave to enter the *Committed* state if all of its causal predecessors are in the *Committed* state. However (as with causal closure) this is insufficient if there are causal cycles. As with causal closure, we introduce a buffer state *PreCommitted*, to which a conclave in the started state can make a transition to, once it is causally closed (Rule (RED AT STPRECMT) in Fig. 7). The transition to the *Committed* state is enabled when all causal predecessors are precommitted (Rule (RED AT PCCOMMIT)). So if a conclave is committed, then it is precommitted and it is closed.

In contrast with causal closure, there is also the possibility of making a transition to an *Aborted* state. A conclave that has not yet precommitted can abort (Rule (RED AT STABORT)). A conclave in the precommitted state can only abort if one of its causal predecessors has aborted (Rule RED AT PCABORT). A summary of the possible state transitions for a conclave is given by:



The transition from the committed state to the aborted state is provided by anticommitment in the lqp(dcu)-calculus, described in Sect. 7.

4.1 Example: Distributed Transactions.

Ignoring aspects of locking for now, we identify a distributed transaction [16] with a collection of conclaves. There is a parent conclave for the start of the transaction at the original site. This transaction invokes operations at remote sites; each instance of the transaction at a remote site is represented by a conclave at that remote site, spawned by the parent conclave. Each child conclave accepts a causal dependency on the parent conclave, and vice versa. At the conclusion of the transaction, the parent conclave executes a two-phase commit protocol:

(1) The parent *conclave* acts as an administrator for the protocol. It contacts each child conclave to induce the latter to enter the *PreCommitted* state, and determines if any children have failed.
(2) If all children have entered the *PreCommitted* state, "evidence" of this is gathered by the administrator and transmitted to the children. The latter use this evidence to enter the *Committed* state. Otherwise the administrator aborts and induces the remaining children to abort (enter the *Aborted* state).

$$\frac{C_1 \textbf{ logged } S_1 \quad C_2 \textbf{ logged } S_2 q \quad S_1 \cap S_2 = \{\}}{(C_1 \mid C_2) \textbf{ logged } S_1 \cup S_2} \text{ (WF PAR)}$$

$$\frac{C \textbf{ logged } S}{(\mathsf{new}\ n;\ C) \textbf{ logged } S - \{n\}} \text{ (WF NEW)} \qquad c\{\!\{L\}\!\} \textbf{ logged } \{c\} \text{ (WF CONJ)} \qquad c\{P\} \textbf{ logged } \{\} \text{ (WF PROC)}$$

Fig. 8. Well-formedness Rules for lqp(dc)-calculus

4.2 Example: Split-Join Transactions.

Such transactions support a split operation that breaks a transaction into two transactions, and a join operation that does the converse [34]. We can define a *split* operation using the operation for "forking" code that executes within another conclave, as well as the operation for initializing the log of a newconclave:

$$(\mathsf{split}\ c\ \mathsf{in}\ P_2;\ P_1) \equiv (\mathsf{new}\ c;\ (c\{\mathsf{loginit};\ P_2\} \mid P_1))$$

Then we have the following execution:

$$\begin{aligned} c_1\{\mathsf{split}\ c\ \mathsf{in}\ P_2;\ P_1\} &\equiv c_1\{\mathsf{new}\ c_2;\ c_2\{\mathsf{loginit};\ P_2\} \mid P_1\} \\ &\equiv \mathsf{new}\ c_2;\ c_1\{c_2\{\mathsf{loginit};\ P_2\} \mid P_1\} \\ &\equiv \mathsf{new}\ c_2;\ (c_1\{c_2\{\mathsf{loginit};\ P_2\}\} \mid c_1\{P_1\}) \\ &\longrightarrow \mathsf{new}\ c_2;\ (c_2\{\mathsf{loginit};\ P_2\} \mid c_1\{P_1\}) \\ &\longrightarrow \mathsf{new}\ c_2;\ (c_2\{\!\{\mathsf{true}\}\!\} \mid c_2\{P_2\} \mid c_1\{P_1\}). \end{aligned}$$

The join operation is provided by having the conclaves for the two transactions become mutually dependent on each other.

5 Correctness

We have two notions of well-formedness for processes; one is a simple syntactic condition that every conclave have exactly one log, while the other is a more sophisticated condition on the consistency of the logs. The latter consistency rules check that certain antecedents hold in the logs if a particular form of log entry is present. For example, if a conclave has a log entry recording that it has committed, then all of its causal predecessors must have committed or must be prepared to

$$\frac{\vdash \overline{C}, C_1, C_2}{\vdash \overline{C}, (C_1 \mid C_2)} \text{ (CONS CTXT PAR)} \qquad \frac{n \notin fn(\overline{C}) \quad \vdash \overline{C}, C}{\vdash \overline{C}, (\mathsf{new}\ n;\ C)} \text{ (CONS CTXT NEW)}$$

$$\frac{\exists S.\ (C_1 \mid \cdots \mid C_k)\ \textbf{logged}\ S \quad \forall i \in \{1,\ldots,k\}.\ (C_1 \mid \cdots \mid C_k) \vdash C_i}{\vdash \overline{C_k}} \text{ (CONS CTXT LOGS)}$$

$$\underset{\text{(CONS PROC)}}{C \vdash c\{P\}} \qquad \underset{\text{(CONS CONJ)}}{\frac{C \vdash c\{\!\{L_1\}\!\} \quad C \vdash c\{\!\{L_2\}\!\}}{C \vdash c\{\!\{L_1 \wedge L_2\}\!\}}} \qquad \underset{\text{(CONS TRUE)}}{C \vdash c\{\!\{\mathsf{true}\}\!\}} \qquad \underset{\text{(CONS CAUSAL PRED)}}{C \vdash c\{\!\{c' \to c\}\!\}}$$

$$\underset{\text{(CONS PRECLOSED)}}{C \vdash c\{\!\{PreClosed\}\!\}} \qquad \underset{\text{(CONS CLOSED)}}{\frac{\begin{array}{c}\forall c' \in (fn(C) - S).\ C \not\models c' \Rightarrow c \\ \forall c' \in S.\ (C \models c' \Rightarrow c \text{ and } C \models c'\{\!\{PreClosed\}\!\})\end{array}}{C \vdash c\{\!\{Closed(S)\}\!\}}}$$

Fig. 9. Log Consistency Rules for lqp(dc)-calculus

$$\underset{\text{(CONS PRECMT SIMPLE)}}{\frac{C \not\models c\{\!\{Aborted\}\!\}}{C \vdash c\{\!\{PreCommitted\}\!\}}} \qquad \underset{\text{(CONS ABORTED SIMPLE)}}{\frac{C \not\models c\{\!\{PreCommitted\}\!\}}{C \vdash c\{\!\{Aborted\}\!\}}}$$

$$\frac{\begin{array}{c}C \models c\{\!\{Aborted\}\!\} \\ C \models c' \Rightarrow c \quad C \models c'\{\!\{Aborted\}\!\} \quad C \not\models c'\{\!\{PreCommitted\}\!\}\end{array}}{C \vdash c\{\!\{PreCommitted\}\!\}} \text{ (CONS PRECMT PREDABT)}$$

$$\frac{\begin{array}{c}C \models c\{\!\{Closed(S)\}\!\} \\ \forall c' \in S.\ C \models c'\{\!\{PreCommitted\}\!\} \text{ and } C \not\models c'\{\!\{Aborted\}\!\}\end{array}}{C \vdash c\{\!\{Committed\}\!\}} \text{ (CONS COMMITTED)}$$

$$\frac{\begin{array}{c}C \models c\{\!\{PreCommitted\}\!\} \\ C \models c' \Rightarrow c \quad C \models c'\{\!\{Aborted\}\!\} \quad C \not\models c'\{\!\{PreCommitted\}\!\}\end{array}}{C \vdash c\{\!\{Aborted\}\!\}} \text{ (CONS ABORTED PREDABT)}$$

Fig. 10. Log Consistency Rules for lqp(dc)-calculus (cont'd)

commit. The well-formedness and log consistency conditions are enforced by the following judgement forms:

C **logged** S	Well-formed conclave	Fig. 8
$\vdash \overline{C}$	Consistent Eval Context	Fig. 9 and 10
$C \vdash C'$	Log consistency	Fig. 9 and 10

The individual log consistency rules are provided in Fig. 9 and 10. Log consistency is denoted by the judgement form $C \vdash C'$. Since evaluation takes place inside an evaluation context, there are also rules in Fig. 9 for checking the log consistency of the evaluation context, with conclusions of the form $\vdash \overline{C}$, where $\overline{C}$ is a multiset of network expressions. These rules break down evaluation contexts until all logs are exposed, and then check (using the remaining rules) that each log is consistent with the other logs.

The (CONS CAUSAL PRED) rule requires that only a conclave can record its causal predecessors; this cannot be done by arbitrary third parties. The (CONS CLOSED) rule for causal closure checks that the set S contains all and only the causal predecessors of c. Furthermore every causal predecessor c' must be prepared to be causally closed (the latter condition is explained in Sect. 3). This rule also requires that all of the logs for the conclaves in S are present in the context C. This ensures that the transitive closure of the set of causal predecessors of c cannot be enlarged with the addition of another log, since the logs of c and its predecessors have enumerated all conclaves in the transitive closure.

Now, turning to Fig. 10, the (CONS PRECMT) rules ensure consistency of precommitment log entries. The (CONS PRECMT SIMPLE) rule handles the simplest case, when a conclave is not aborted. The check for the absence of a log entry for abortion therefore requires that the log for the conclave c be present in the context. The (CONS PRECMT PREDABT) rule handles the case where a conclave may be both precommitted and aborted; this can be the case when a predecessor of the conclave is aborted.

The (CONS COMMITTED) rule requires that the conclave c be causally closed, and that each of its causal predecessors be prepared to commit (as explained in Sect. 4). Some of these predecessors may actually have committed, but precommitment is all that is required for consistency. For consistency we also require that none of the causal predecessors of c be aborted, requiring that the logs of all of these predecessors be in the context of the consistency check.

There are two consistency rules for abortion. The (CONS ABORTED SIMPLE) rule handles the simplest case, when a conclave is not precommitted and therefore there are no restrictions on abortion. The (CONS ABORTED PREDABT), analogous to the (CONS PRECMT PREDABT) rule for precommitment, handles the case where a precommitted conclave's abortion is justified by an abort log entry for one of its causal predecessors.

The (CONS PRECMT PREDABT) rule checks for the existence of at least one aborted, non-precommitted conclave. This check stops us from writing down a consistent network that reduces to an inconsistent one. For example, without this restriction the following network would be consistent.

$$
\begin{aligned}
&c_1\{\!\{\mathit{PreCommitted} \wedge \mathit{PreClosed} \wedge \mathit{Closed}(\{c_1,c_2,c_3\}) \wedge \mathit{Aborted}\}\!\}\\
\mid\, &c_2\{\!\{\mathit{PreCommitted} \wedge \mathit{PreClosed} \wedge \mathit{Closed}(\{c_1,c_2,c_3\}) \wedge \mathit{Aborted}\}\!\}\\
\mid\, &c_3\{\!\{\mathit{PreCommitted} \wedge \mathit{PreClosed} \wedge \mathit{Closed}(\{c_1,c_2,c_3\})\}\!\}
\end{aligned}
$$

The conclave c_3 could commit, even though its other predecessors are aborted. So, if this network was consistent we would have a consistent network reducing to a inconsistent one.

We now consider the correctness of the semantics, and in particular the correctness of the log append rules.

Definition 5.1 (Log Consistency) *A network C is* log-consistent *if* $\vdash C$ *is derivable using the derivation rules in Fig. 9 and 10. We sometimes write this as* $\vdash C$.

Lemma 5.1 *Suppose the following hold:*

(1) $C, c \models (\overline{v}) \xrightarrow{\textit{rule-name}} (\mathsf{new}\ \overline{n};\ L)$.
(2) $\vdash \overline{C}, C, C'$ *where* $(\mathit{fn}(\overline{C}) \cup \mathit{fn}(C) \cup \mathit{fn}(C')) \cap \{\overline{n}\} = \{\}$.

Then $(C \mid C'), c \models (\overline{v}) \xrightarrow{\textit{rule-name}} (\mathsf{new}\ \overline{n};\ L)$. *In other words, a log append rule is not disabled by the addition of logs* C' *that are consistent with the logs C already enabling the rule.*

Proof Sketch: By induction on the additional collection of logs C'. The base case considers all possible combinations of log entry and log append rule. □

The following key lemma verifies that the log append rules preserve log consistency.

Lemma 5.2 *Suppose the following hold:*

(1) $C \equiv (C'_0 \mid c\{\!\{L_0\}\!\})$.
(2) $C, c \models (\overline{v}) \xrightarrow{\textit{rule-name}} (\mathsf{new}\ \overline{n'};\ L'_0)$.
(3) $\vdash \overline{C}, C$ *where* $(\mathit{fn}(\overline{C}) \cup \mathit{fn}(C)) \cap \{\overline{n'}\} = \{\}$.

Then $\vdash \overline{C}, (C'_0 \mid c\{\!\{L_0 \wedge L'_0\}\!\})$. *In other words, if a collection of logs C is consistent with the logs in* $\{\overline{C}\}$ *and moreover enables a log append rule, then the collection of logs resulting from executing this log append is still consistent with the logs in* $\{\overline{C}\}$.

Proof: We use the assumption $\vdash \overline{C}, (C'_0 \mid c\{\!\{L_0\}\!\})$, i.e., that the logs are consistent before the log append, to show that the logs remain consistent after the log append: $\vdash \overline{C}, (C'_0 \mid c\{\!\{L_0 \wedge L'_0\}\!\})$. We verify the lemma by induction on the size of the multiset $\{\overline{C}\}$. The inductive step is for the case where $\overline{C} = \overline{C'}, C'$. We already have that $\vdash \overline{C'}, C', C$. Therefore by Lemma 5.1 we have that

$$(C \mid C'), c \models (\overline{v}) \xrightarrow{\textit{rule-name}} (\mathsf{new}\ \overline{n'};\ L'_0),$$

where $(C \mid C') \equiv (C' \mid C'_0 \mid c\{\!\{L_0\}\!\})$. Since $\vdash \overline{C'}, (C \mid C')$ by Rule (CON CTXT PAR), we may apply the induction hypothesis with $\overline{C'}$ and $(C \mid C')$ to obtain $\vdash \overline{C'}, (C' \mid C'_0 \mid c\{\!\{L_0 \wedge L'_0\}\!\})$, so we must have a subderivation for $\vdash \overline{C'}, C', (C'_0 \mid c\{\!\{L_0 \wedge L'_0\}\!\})$.

It remains to consider the base cases, one for each log append rule, where we have a derivation for $\vdash C$. The details are provided in a technical report [35]. $\square$

Lemma 5.3 *Suppose the following hold:*

(1) $\vdash \overline{C}, C$.
(2) $C \equiv \mathbb{E}[C_0]$ *for some evaluation context* $\mathbb{E}[\ \cdot\ ]$.

Then for some $\overline{n'}$ *and* $\overline{C'}$*, we have that* $\mathbb{E}[\ \cdot\ ] \equiv (\mathsf{new}\ \overline{n'};\ ([\ \cdot\ ] \mid \prod \overline{C'}))$ *and* $\vdash \overline{C}, \overline{C'}, C_0$. *In other words, any consistency derivation gives rise to an equivalent derivation (i.e., same conclusion) where an evaluation context in one of the network descriptions in the conclusion is verified consistent as the last step in the derivation (equivalently, is decomposed from the root).*

Theorem 1 (Preservation of Log Consistency) *If* $\vdash \overline{C}, C$ *and* $C \longrightarrow C'$*, then* $\vdash \overline{C}, C'$.

Proof: By induction on the derivation for the reduction step:

Case (RED CONG)**:** We have $C \equiv \mathbb{E}[C_0]$ and $C' \equiv \mathbb{E}[C'_0]$ and $C_0 \longrightarrow C'_0$. Then by Lemma 5.3 we have that $\mathbb{E}[\ \cdot\ ] \equiv (\mathsf{new}\ \overline{n'};\ ([\ \cdot\ ] \mid \prod \overline{C'}))$ and $\vdash \overline{C}, \overline{C'}, C_0$. By the induction hypothesis applied to $C_0 \longrightarrow C'_0$, we have that $\vdash \overline{C}, \overline{C'}, C'_0$, and therefore $\vdash \overline{C}, \mathbb{E}[C'_0]$ by applications of (CONS PAR) and (CONS NEW).
Case (RED APPEND)**:** This is an easy consequence of Lemma 5.2.

Since the remaining reduction rules have no effect on the logs, we are done. $\square$

Corollary 5.1 *For the lqp(dc)-calculus, if* $\vdash C_1$ *and* $C_1 \stackrel{*}{\longrightarrow} C_2$*, then* $\vdash C_2$*.*

This can be viewed as a form of subject reduction. Typically subject reduction proofs verify that certain static properties remain invariant during program execution, and this invariant is then used to confirm some property of program execution. For example subject reduction may be used to confirm progress: a program does not get stuck because of type errors. In our approach the invariants (represented by the consistency of the logs) and the properties that they enforce are completely application dependent. For example, the consistency of logs with commitment, abortion and dependencies could be used to verify that any trace of the execution of a collection of processes is equivalent to a trace where the traces of the processes of the uncommitted conclaves have been erased. In particular none of the remaining processes relies on the results of a process that has aborted or may potentially abort.

6 Extended Example: The ATF-Calculus

The atf-calculus [36] is a process calculus with atomic failure as its central organizing principle. For a committed transaction in the atf-calculus, it is guaranteed to have only received messages from other committed transactions. So the "effects" of uncommitted transactions, the messages they sent, are guaranteed to be ignored by committed transactions. This example demonstrates how particular patterns of programming with atomic failures can be constructed atop the primitives of the lqp(dc)-calculus.

6.1 Definition of the atf-Calculus

The syntax of the atf-calculus is given by:

$$\begin{array}{rcl}
A \in \mathit{Process} & ::= & \mathsf{stop} \;\mid\; \mathsf{repeat}\; A \;\mid\; (A_1 \mid A_2) \;\mid\; \mathsf{new}\; n;\; A \\
& \mid & \mathsf{send}\; v!\langle\overline{v_k}\rangle \;\mid\; \mathsf{receive}\; n?\overline{x_k};\; A \\
& \mid & \mathsf{receive\ committed}\; n?\overline{x_k};\; A \\
& \mid & \mathsf{prepare} \;\mid\; \mathsf{abort} \;\mid\; \mathsf{commit} \\
T \in \mathit{Trans} & ::= & t\langle A\rangle \;\mid\; t\langle\!\langle L\rangle\!\rangle \;\mid\; (T_1 \mid T_2) \;\mid\; \mathsf{new}\; n;\; T \\
L \in \mathit{Log} & ::= & \mathsf{prepare} \;\mid\; \mathsf{abort} \;\mid\; \mathsf{commit} \;\mid\; t_1\langle \mathsf{send}\; v!\langle\overline{v_k}\rangle\rangle \;\mid\; t_1 \rightarrow t_2 \\
& \mid & \mathsf{true} \;\mid\; L_1 \wedge L_2
\end{array}$$

The semantics of the atf-calculus is presented in Fig. 11 and Fig. 12.

We assume similar structural equivalence rules as for the lqp(dc)-calculus. So for example both parallel composition and log conjunction are associative and commutative. Besides the usual operations for stopped processes, replication, parallel composition, new name generation, and message passing, there are operations for aborting and committing transactions. Commitment is similar to the *early prepare commit protocol* [37], whereby participants prepare for commitment and an administrator then decides if the participants should commit. It is also similar to completion in the lqp(dc)-calculus, except that the latter has more freedom for the application to decide when to enter the phases of the commitment protocol. The prepare operation puts a participant into the prepared state, recorded by a log entry of the form $t\langle\!\langle \mathsf{prepare} \rangle\!\rangle$. A participant can commit if all of its causal predecessors are either committed or prepared. A participant can abort if any of its causal predecessors has aborted. A participant can autonomously abort any time before it enters the prepared state.

As with the lqp(dc)-calculus, there are logs of the form $t\langle\!\langle \ldots \rangle\!\rangle$, at most one per transaction. There are five types of log entries that are actually stored in the log. Three of these log entry types record if a transaction is in the prepared, committed or aborted state. The fourth log entry type, $t_1\langle \mathsf{send}\ v!\langle \overline{v_k} \rangle\rangle$, records that (a process in) the current transaction received a message sent by (a process in) the transaction t_1. This log entry type is useful for undoing the effects of an aborted transaction, retransmitting any messages that the transaction had received before aborting. This is done by the (RED ATF UNDO) rule in Fig. 11. The fifth log entry type, $t_1 \to t_2$, records immediate causal dependencies. Initially such an entry is added to the log when a message receipt is logged establishing a dependence of t_2 on t_1. Since message undo removes the log of message receipt, a separate log record of the dependency is required to ensure log consistency, since an aborted prepared transaction requires an aborted predecessor which is not prepared and which might not be an immediate predecessor.

The semantics for the atf-calculus are specified using various judgement forms:

$A_1 \equiv A_2$	Process equivalence	Not shown
$L_1 \equiv L_2$	Log equivalence	Not shown
$T_1 \equiv T_2$	Transaction equivalence	Not shown
$T_1 \longrightarrow T_2$	Computation	Fig. 11
$T \models t\langle\!\langle L \rangle\!\rangle$	Log query	Some in Fig. 12

The reflexive transitive closure of the computation relation is denoted by $T_1 \stackrel{*}{\longrightarrow} T_2$. The rules for the derived causality relation are given in Fig. 12, with the base case given by an immediate predecessor link in the logs (Rule (PRED ATF CAUSAL HYP)).

There are two operations for receiving messages. The first operation corresponds to receiving a message and accepting a causal dependency on the sending transaction.

$$\mathbb{E}[\,\cdot\,] ::= [\,\cdot\,] \;\mid\; (\mathbb{E}[\,\cdot\,] \mid T) \;\mid\; \mathsf{new}\ n;\ \mathbb{E}[\,\cdot\,]$$

$$\frac{T_1 \equiv \mathbb{E}[T_1'] \quad T_1' \longrightarrow T_2' \quad T_2 \equiv \mathbb{E}[T_2']}{T_1 \longrightarrow T_2} \quad \text{(RED ATF CONG)}$$

$$\frac{T_1 = t_1\langle \mathsf{send}\ v!\langle v_1,\ldots,v_k\rangle\rangle \quad T_2 = t_2\langle \mathsf{receive}\ v?x_1,\ldots,x_k;\ P\rangle \quad t_2\langle\!\langle L\rangle\!\rangle \not\models t_2\langle\!\langle \mathsf{prepare}\rangle\!\rangle}{T_1 \mid T_2 \mid t_2\langle\!\langle L\rangle\!\rangle \longrightarrow t_2\langle\{\overline{v_k}/\overline{x_k}\}P\rangle \mid t_2\langle\!\langle L \wedge T_1 \wedge (t_1 \to t_2)\rangle\!\rangle} \quad \text{(RED ATF RECEIVE)}$$

$$\frac{T_1 = t_1\langle \mathsf{send}\ v!\langle v_1,\ldots,v_k\rangle\rangle \quad T_2 = t_2\langle \mathsf{receive\ committed}\ v?x_1,\ldots,x_k;\ P\rangle \quad t_1\langle\!\langle L_1\rangle\!\rangle \models t_1\langle\!\langle \mathsf{commit}\rangle\!\rangle}{T_1 \mid T_2 \mid t_1\langle\!\langle L_1\rangle\!\rangle \mid t_2\langle\!\langle L_2\rangle\!\rangle \longrightarrow t_2\langle\{\overline{v_k}/\overline{x_k}\}P\rangle \mid t_1\langle\!\langle L_1\rangle\!\rangle \mid t_2\langle\!\langle L_2 \wedge T_1\rangle\!\rangle}$$
(RED ATF RECVCOMM)

$$\frac{T = t'\langle \mathsf{send}\ v!\langle v_1,\ldots,v_k\rangle\rangle}{t\langle\!\langle L \wedge T \wedge \mathsf{abort}\rangle\!\rangle \longrightarrow t\langle\!\langle L \wedge \mathsf{abort}\rangle\!\rangle \mid T} \quad \text{(RED ATF UNDO)}$$

$$\frac{t\langle\!\langle L\rangle\!\rangle \not\models t\langle\!\langle \mathsf{prepare}\rangle\!\rangle}{t\langle \mathsf{abort}\rangle \mid t\langle\!\langle L\rangle\!\rangle \longrightarrow t\langle\!\langle L \wedge \mathsf{abort}\rangle\!\rangle} \quad \text{(RED ATF ABORT)}$$

$$\frac{t\langle\!\langle L\rangle\!\rangle \not\models t\langle\!\langle \mathsf{abort}\rangle\!\rangle}{t\langle \mathsf{prepare}\rangle \mid t\langle\!\langle L\rangle\!\rangle \longrightarrow t\langle\!\langle L \wedge \mathsf{prepare}\rangle\!\rangle} \quad \text{(RED ATF PREPARE)}$$

$$\frac{t\langle\!\langle L\rangle\!\rangle \not\models t\langle\!\langle \mathsf{commit}\rangle\!\rangle \quad t\langle\!\langle L\rangle\!\rangle \models t\langle\!\langle \mathsf{prepare}\rangle\!\rangle \quad (T \mid t\langle\!\langle L\rangle\!\rangle) \models t' \Rightarrow t,\ t'\langle\!\langle \mathsf{abort}\rangle\!\rangle}{t\langle \mathsf{abort}\rangle \mid t\langle\!\langle L\rangle\!\rangle \mid T \longrightarrow t\langle\!\langle L \wedge \mathsf{abort}\rangle\!\rangle \mid T} \quad \text{(RED ATF PREPABT)}$$

$$\frac{T = (t\langle\!\langle L\rangle\!\rangle \mid T') \quad \exists S.\ t \in S \text{ and } \forall t' \in S.\ (T \models t'\langle\!\langle \mathsf{prepare}\rangle\!\rangle) \text{ and } (\forall t''.\ T \models t'\langle\!\langle t'' \to t'\rangle\!\rangle \text{ implies } t'' \in S)}{t\langle \mathsf{commit}\rangle \mid t\langle\!\langle L\rangle\!\rangle \mid T' \longrightarrow t\langle\!\langle L \wedge \mathsf{commit}\rangle\!\rangle \mid T'}$$
(RED ATF COMMIT)

Fig. 11. Semantics of ATF Calculus: Computation Rules

$$\frac{T \models t\langle\!\langle t' \to t\rangle\!\rangle}{T \models t' \Rightarrow t} \quad \text{(PRED ATF CAUSAL HYP)}$$

$$T \models t \Rightarrow t \quad \text{(PRED ATF CAUSAL REFL)}$$

$$\frac{T \models t_1 \Rightarrow t_2 \quad T \models t_2 \Rightarrow t_3}{T \models t_1 \Rightarrow t_3} \quad \text{(PRED ATF CAUSAL TRANS)}$$

Fig. 12. Semantics of ATF Calculus: Causality Rules

A transaction cannot in general receive new messages if it has entered the prepared state, since that might introduce new causal dependencies on transactions that could then abort, invalidating any earlier decision to commit. This is enforced by the antecedent in the (RED ATF RECEIVE) rule. The second message receive operation restricts received messages to those that were sent by committed transactions (so a process can isolate itself from the effects of uncommitted transactions). Once a transaction has entered the prepared state, it can only receive messages from other committed transactions. This prevents further causal dependencies from being introduced while committing a transaction, and prevents a committed transaction from gaining a causal dependency from an aborted transaction. This is handled by the (RED ATF RECVCOMM) rule that only receives messages from committed transactions. Both forms of the message receive operation keep a log of the message to allow the message receive to be undone if the receiver subsequently aborts. The first form of receive also records a causal dependency of the receiver on the sender.

The (RED ATF UNDO) rule allows a message receipt to be undone, using the logs, once the receiving process has aborted. Undoing a message receipt involves putting that message back on the channel from which it was removed. The causal dependency that may be recorded in the log after the transaction originally received the message is left in the log, since it may be required for a causal chain that allows a prepared transaction to abort. The (RED ATF ABORT) and (RED ATF PREPARE) rules allow a transaction to abort or enter the prepare state, respectively, provided that it has not yet entered any other state. The (RED ATF PREPABT) rule allows a transaction to abort while it is in the prepare state, if it has not yet committed and one of its causal predecessors has aborted.

Finally the heart of the atf-calculus is the (RED ATF COMMIT) rule, that allows a transaction to commit, provided that all of its causal predecessors are in the prepared state (some of them may have already committed). This condition is checked by ensuring that the transitive closure of the predecessor relation rooted at the transaction t is well-defined. The definition of transitive closure requires that the logs for all predecessors be available in the context, and that all such predecessors be in the prepare state.

6.2 Correctness Overview

The atf-calculus' key property is that messages sent to transactions, which then abort, are not "lost". The proof that this property holds is easily verified, as it follows from the way in which the reduction rules are written:

Lemma 6.1 *If* $T \equiv (T_1 \mid t_1\langle \mathsf{send}\ v!\langle v_1,\ldots,v_k\rangle\rangle)$ *and* $T \xrightarrow{*} T'$ *then either:*

- T' *contains an unaborted transaction with the log entry* $(t_1\langle \mathsf{send}\ v!\langle v_1,\ldots,v_k\rangle\rangle)$, *or*
- *there exists some* T'' *and* T_1'' *such that* $T' \xrightarrow{*} T''$ *and* $T'' \equiv (T_1'' \mid t_1\langle \mathsf{send}\ v!\langle v_1,\ldots,v_k\rangle\rangle)$.

Proof: If the reduction from T to T' does not involve the send $v!\langle\overline{v_k}\rangle$ action then it must still be an unguarded part of the process, so $T' \equiv T_1' \mid t_1\langle \mathsf{send}\ v!\langle\overline{v_k}\rangle\rangle$.

If on the other hand, the reduction did involve the send $v!\langle\overline{v_k}\rangle$ action it must then have used the (RED ATF RECEIVE) rule or the (RED ATF RECVCOMM) rule, as these are the only two rules that involve a send $v!\langle\overline{v_k}\rangle$ action. Both of these rules add a $t_1\langle\mathsf{send}\ v!\langle\overline{v_k}\rangle\rangle$ entry into a log. We now have two further cases. If the transaction has not aborted, we are done. If the particular transaction in question has aborted then its log contains an abort entry. The (RED ATF UNDO) rule can then be applied to T' to show that $T' \longrightarrow T_1' \mid t_1\langle\mathsf{send}\ v!\langle\overline{v_k}\rangle\rangle$.

If the (RED ATF UNDO) was applied to resend the message, then we have returned to our first case and we start again.

$\square$

So if a message sent by a transaction t_1 is received by a transaction t_2, then as long as t_2 is not aborted, the latter's logs will record the receipt of the message. But if t_2 does abort, then eventually the message will appear again on the output channel its receipt having been undone by the semantics of the atf-calculus. As we are using asynchronous communication where an output message never guards a process, this rebroadcasting of the output cancels the effect of the aborting process accepting the communication from the outputting process.

However, this property is of little use if a transaction's logs are inconsistent. To this end, correctness for the atf-calculus is defined in much the same way as for the lqp(dc)-calculus, in terms of log consistency. The well-formedness and log consistency conditions for the atf-calculus are enforced by the following judgement forms:

T **logged** S	Well-formed transaction	Not shown
$\vdash \overline{T}$	Consistent Eval Context	Fig. 13
$T \vdash T'$	Log consistency	Fig. 13

Definition 6.1 *A transaction T is* log-consistent if $\vdash T$ *is derivable using the derivation rules in Figure 13.*

These rules are similar to the consistency rules for commitment in the lqp(dc)-calculus. The last five rules in Fig. 13 correspond to the rules in Fig. 10.

The lqp(dc)-calculus is intended as a "base language" atop which other more high-level languages can be designed and implemented. The atf-calculus is an example of this. As such, we can make use of the correctness results for the lqp(dc)-calculus to leverage a correctness proof for the atf-calculus. We do this by first giving a translation from the atf-calculus to the lqp(dc)-calculus, that preserves and reflects log consistency. We then show operational correspondence [38]: a reduction by a trans-

$$\frac{\vdash \overline{T}, T_1, T_2}{\vdash \overline{T}, (T_1 \mid T_2)} \text{ (CONS ATF CTXT PAR)} \qquad \frac{n \notin fn(\overline{T}) \quad \vdash \overline{T}, T}{\vdash \overline{T}, (\mathsf{new}\ n;\ T)} \text{ (CONS ATF CTXT NEW)}$$

$$\frac{\exists S.\ (T_1 \mid \cdots \mid T_k)\ \textbf{logged}\ S \qquad \forall i \in \{1, \ldots, k\}.\ \prod \overline{T_k} \vdash T_i}{\vdash \overline{T_k}} \text{ (CONS ATF CTXT LOGS)}$$

$$T \vdash t\langle A \rangle \text{ (CONS ATF PROC)} \qquad \frac{T \vdash t\langle\!\langle L_1 \rangle\!\rangle \quad T \vdash t\langle\!\langle L_2 \rangle\!\rangle}{T \vdash t\langle\!\langle L_1 \wedge L_2 \rangle\!\rangle} \text{ (CONS ATF CONJ)} \qquad T \vdash t\langle\!\langle \mathsf{true} \rangle\!\rangle \text{ (CONS ATF TRUE)}$$

$$T \vdash t\langle\!\langle t' \to t \rangle\!\rangle \text{ (CONS ATF PRED)} \qquad \frac{T \models t\langle\!\langle t' \to t \rangle\!\rangle}{T \vdash t\langle\!\langle t'\langle \mathsf{send}\ v!\langle \overline{v_k} \rangle \rangle \rangle\!\rangle} \text{ (CONS ATF MSG)} \qquad \frac{T \models t'\langle\!\langle \mathsf{commit} \rangle\!\rangle}{T \vdash t\langle\!\langle t'\langle \mathsf{send}\ v!\langle \overline{v_k} \rangle \rangle \rangle\!\rangle} \text{ (CONS ATF MSG COM)}$$

$$\frac{T \not\models t\langle\!\langle \mathsf{abort} \rangle\!\rangle}{T \vdash t\langle\!\langle \mathsf{prepare} \rangle\!\rangle} \text{ (CONS ATF PREP SIMPLE)} \qquad \frac{T \not\models t\langle \mathsf{prepare} \rangle}{T \vdash t\langle\!\langle \mathsf{abort} \rangle\!\rangle} \text{ (CONS ATF ABORT SIMPLE)}$$

$$\frac{\forall t'.\ (T \models t' \Rightarrow t) \text{ implies } (T \models t'\langle\!\langle \mathsf{prepare} \rangle\!\rangle \text{ and } T \not\models t'\langle\!\langle \mathsf{abort} \rangle\!\rangle)}{T \vdash t\langle\!\langle \mathsf{commit} \rangle\!\rangle} \text{ (CONS ATF COMMIT)}$$

$$\frac{T \models t\langle\!\langle \mathsf{abort} \rangle\!\rangle \quad T \models t' \Rightarrow t \qquad T \models t'\langle\!\langle \mathsf{abort} \rangle\!\rangle \quad T \not\models t'\langle\!\langle \mathsf{prepare} \rangle\!\rangle}{T \vdash t\langle\!\langle \mathsf{prepare} \rangle\!\rangle} \text{ (CONS ATF PREP PREDABT)}$$

$$\frac{T \models t\langle \mathsf{prepare} \rangle \quad T \models t' \Rightarrow t \qquad T \models t'\langle\!\langle \mathsf{abort} \rangle\!\rangle \quad T \not\models t'\langle\!\langle \mathsf{prepare} \rangle\!\rangle}{T \vdash t\langle\!\langle \mathsf{abort} \rangle\!\rangle} \text{ (CONS ATF ABORT PREDABT)}$$

Fig. 13. Log Consistency Rules for the atf-calculus

action in the atf-calculus can always be matched by a number of reductions of that transaction's translation. Theorem 1 verifies that reductions in the lqp(dc)-calculus preserve log consistency. Therefore if the original transaction in the atf-calculus is consistent, then any terms that the transaction's translation reduces to will also be consistent. So, by way of these operational correspondence and encoding results, we verify that reduction in the atf-calculus preserves log consistency.

6.3 Translating the ATF-calculus to the lqp(dc)-calculus

The translation of the atf-calculus into the lqp(dc)-calculus is provided in Fig. 14. The aim of this translation is to allow us to reason about the correctness of the atf-calculus in terms of the lqp(dc)-calculus. As such the translation does not provide a particularly efficient implementation of the atf-calculus but it does preserve consistency. This translation is specified using various meta-functions:

$\mathcal{T}[\![T]\!]$	Top-level translation of transactions to conclaves
$\mathcal{T}[\![T]\!]\eta$	Translation of transactions to conclaves
$\mathcal{T}[\![L]\!]t$	Translation of log entries to undo code
$\mathcal{L}[\![L]\!]S\ t$	Translation of atf-calculus logs to lqp(dc)-calculus logs
$\mathcal{P}[\![A]\!]t$	Translation of ATF-calculus processes to the lqp(dc)-calculus

We make the following simplifying assumption about the structure of transactions for the translation. We assume that names are separated into two disjoint sets: transaction names and channel names (with renaming respecting this set membership; a renaming of a bound name must be to a name in the same set). A transaction log $t\langle\!\langle \ldots \rangle\!\rangle$ requires that t be a transaction name. We assume that a top-level transaction network description has the form $\mathsf{new}\ \bar{t};\ T$, for some T, where $\{\bar{t}\}$ are all transaction names, where the only free transaction names in T are in $\{\bar{t}\}$, and where T cannot bind any transaction names. This assumption simplifies reasoning about the translation.

The translation at the top-level is defined as $\mathcal{T}[\![\mathsf{new}\ \bar{t};\ T]\!]$, where the binder binds all transaction names in T. The translation of transactions $\mathcal{T}[\![T]\!]\eta$ is parameterized by a mapping η from each transaction t to the set of its immediate causal predecessors. This is used to compute the set of causal predecessors S when the translation generates log entries of the form $\mathit{Closed}(S)$.

For a log L, the translation $\mathcal{T}[\![L]\!]t$ constructs a collection of processes that wait for the corresponding transaction to abort and then resend messages that were received by that transaction. The translation $\mathcal{L}[\![L]\!]S\ t$ converts from atf-calculus log entries to lqp(dc)-calculus log entries. The translation $\mathcal{T}[\![T]\!]\eta$ invokes both of the aforesaid translations when applied to a log in the atf-calculus; otherwise the only other interesting case is for processes, where it invokes the translation $\mathcal{P}[\![A]\!]t$ translating from atf-calculus processes to lqp(dc)-calculus processes.

In the latter translation, in the translation of the first message receive operation (the operation that accepts messages from uncommitted transactions), messages are augmented with the conclave identifier of the transaction sending the message, and this is used at the receiver to record the causal dependency between sender and receiver.

For the second message receive operation, the receive-commit operation that only receives messages from committed transactions, the receiving operation polls the input channel until it receives a message that was sent by a transaction that has committed. For this we assume a definable extension of the lqp(dc)-calculus with a logcheck construct, for checking the presence or absence of remote log entries.

This has the computation rules:

$$\frac{c\{\!\{L'\}\!\} \models c\{\!\{L\}\!\}}{(c\{\!\{L'\}\!\} \mid c'\{\mathsf{logcheck}\ c\{\!\{L\}\!\}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\}) \longrightarrow (c\{\!\{L'\}\!\} \mid c'\{P_1\})}$$

(RED CHKLOGTRUE)

$$(c\{\!\{L'\}\!\} \mid c'\{\mathsf{logcheck}\ c\{\!\{L\}\!\}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\}) \longrightarrow (c\{\!\{L'\}\!\} \mid c'\{P_2\})$$

(RED CHKLOGFALSE)

As noted in Sect. 2, we disallow querying for the absence of remote log entries because an implementation would require distributed agreement to avoid race conditions. Therefore the (RED CHKLOGFALSE) rule allows the conditional to pessimistically assume that a remote log entry is not present. The logcheck can be defined in terms of logawait:

$$\begin{aligned}&\mathsf{logcheck}\ c\{\!\{L\}\!\}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \equiv \\ &\qquad \mathsf{new}\ a;\ (\ \mathsf{logawait}\ c\{\!\{L\}\!\};\ \mathsf{receive}\ a?();\ P_1 \\ &\qquad\qquad \mid \mathsf{receive}\ a?();\ P_2 \\ &\qquad\qquad \mid \mathsf{send}\ a!\langle\rangle\ )\end{aligned}$$

Even though the construct may nondeterministically pick the false branch even when the log entry is present, the process in this case simply loops to check the condition again, for another message on the channel. The process that does this polling is defined using the standard encoding of recursive processes in the pi-calculus [24]:

$$\mathsf{new}\ a;\ ((\mathsf{repeat\ receive}\ a?();\ P(a)) \mid (\mathsf{send}\ a!\langle\rangle))$$

where a is a "trigger" channel for forcing invocations of the process. If the process body $P(a)$ wishes to perform a recursive invocation of itself, it sends a message send $a!\langle\rangle$.

6.4 Correctness of the Encoding

To relate log consistency in the atf-calculus and in the lqp(dc)-calculus, we observe that there is a very strong relationship based on erasing processes and only considering logs and contexts:

Definition 6.2 (Process Erasure) *Define $\mathcal{E}[\![T]\!]$ to be the erasure of all processes from a network description. It is the obvious homomorphic extension of:*

$$\mathcal{E}[\![t\langle A\rangle]\!] = t\langle \mathsf{stop}\rangle.$$

$$\mathcal{T}[\![\mathsf{new}\ \overline{t};\ T]\!] = \mathsf{new}\ \overline{t};\ \mathcal{T}[\![T]\!]\{\overline{t} \mapsto \overline{S}| \text{ where } S_i = \{t' \mid \exists T'.\ T \equiv (\mathsf{new}\ \overline{n};\ (T' \mid t\langle\!\langle L \wedge t' \to t \rangle\!\rangle))\}\}$$
$$\text{where } T \text{ has no bound occurrences of transaction names}$$
$$\mathcal{T}[\![t\langle\!\langle L \rangle\!\rangle]\!]\eta = (\mathcal{T}[\![L]\!]t \mid t\{\!\{\mathcal{L}[\![L]\!]S\ t\}\!\}) \text{ where } S = \bigcap\{S' \mid t \in S' \text{ and } \forall t' \in S'.\ \eta(t') \subseteq S'\}$$

$$\mathcal{T}[\![\mathsf{new}\ n;\ T]\!]\eta = \mathsf{new}\ n;\ \mathcal{T}[\![T]\!]\eta \qquad \mathcal{T}[\![T_1 \mid T_2]\!]\eta = (\mathcal{T}[\![T_1]\!]\eta \mid \mathcal{T}[\![T_2]\!]\eta) \qquad \mathcal{T}[\![t\langle A \rangle]\!]\eta = t\{\mathcal{P}[\![A]\!]t\}$$

$$\begin{aligned}
\mathcal{T}[\![t_0\langle \mathsf{send}\ v!\langle\overline{v_k}\rangle\rangle]\!]t &= t\{\mathsf{logawait}\ t_0\{\!\{\mathit{Aborted}\}\!\};\ \mathsf{send}\ v!\langle t_0, \overline{v_k}\rangle\} \\
\mathcal{T}[\![L_1 \wedge L_2]\!]t &= (\mathcal{T}[\![L_1]\!]t \mid \mathcal{T}[\![L_2]\!]t) \\
\mathcal{T}[\![L]\!]t &= t\{\mathsf{stop}\} \quad \text{otherwise} \\
\mathcal{L}[\![\mathsf{prepare}]\!]S\ t &= \mathit{PreCommitted} \wedge \mathit{PreClosed} \\
\mathcal{L}[\![\mathsf{commit}]\!]S\ t &= \mathit{Committed} \wedge \mathit{Closed}(S) \\
\mathcal{L}[\![\mathsf{abort}]\!]S\ t &= \mathit{Aborted} \\
\mathcal{L}[\![(t' \to t)]\!]S\ t &= (t' \to t) \\
\mathcal{L}[\![L_1 \wedge L_2]\!]S\ t &= (\mathcal{L}[\![L_1]\!]S\ t) \wedge (\mathcal{L}[\![L_2]\!]S\ t) \\
\mathcal{L}[\![L]\!]S\ t &= \mathsf{true} \quad \text{otherwise} \\
\mathcal{P}[\![\mathsf{prepare}]\!]t &= \mathsf{logappend}\ \langle\rangle\ \mathsf{with}\ \mathit{AtStPreCommit};\ \mathsf{logappend}\ \langle\rangle\ \mathsf{with}\ \mathit{PreClosed};\ \mathsf{stop}\ \ //\text{etc for other log commands}
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}[\![\mathsf{receive}\ n?\overline{x_k};\ A']\!]t = {} & \mathsf{receive}\ n?(y, \overline{x_k});\ \mathsf{logif}\ t\{\!\{\mathit{PreClosed}\}\!\}\ \mathsf{then}\ \mathsf{send}\ n!\langle y, \overline{x_k}\rangle\ \mathsf{else} \\
& ((\mathsf{logappend}\ \langle y \rangle\ \mathsf{with}\ \mathit{CausalPred};\ \mathcal{P}[\![A']\!]t) \mid (\mathsf{logawait}\ t\{\!\{\mathit{Aborted}\}\!\};\ \mathsf{send}\ n!\langle y, \overline{x_k}\rangle))
\end{aligned}$$

$$\begin{aligned}
\mathcal{P}[\![\mathsf{receive\ committed}\ n?\overline{x_k};\ A']\!]t = {} & \mathsf{new}\ \mathit{trig};\ ((\mathsf{repeat\ receive}\ \mathit{trig}?();\ \mathit{LOOP}) \mid \mathsf{send}\ \mathit{trig}!\langle\rangle) \\
& \text{where } \mathit{LOOP} = \mathsf{receive}\ n?(y, \overline{x_k});\ \mathsf{logcheck}\ y\{\!\{\mathit{Committed}\}\!\}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \\
& \text{and } P_1 = (\mathcal{P}[\![A']\!]t \mid (\mathsf{logawait}\ t\{\!\{\mathit{Aborted}\}\!\};\ \mathsf{send}\ n!\langle y, \overline{x_k}\rangle)) \\
& \text{and } P_2 = (\mathsf{send}\ n!\langle y, \overline{x_k}\rangle \mid \mathsf{send}\ \mathit{trig}!\langle\rangle)
\end{aligned}$$

$$\mathcal{P}[\![\mathsf{new}\ n;\ A]\!]t = \mathsf{new}\ n;\ \mathcal{P}[\![A]\!]t \qquad \mathcal{P}[\![A_1 \mid A_2]\!]t = (\mathcal{P}[\![A_1]\!]t \mid \mathcal{P}[\![A_2]\!]t) \qquad \mathcal{P}[\![\mathsf{send}\ v!\langle\overline{v_k}\rangle]\!]t = \mathsf{send}\ v!\langle t, \overline{v_k}\rangle$$

Fig. 14. Translation of the atf-calculus

Lemma 6.2 *For any transaction T, we have $\vdash T$ if and only if $\vdash \mathcal{E}[\![T]\!]$.*

A similar definition and lemma are possible for networks in the lqp(dc)-calculus, $\mathcal{E}[\![C]\!]$. The following theorem verifies that the encoding of the atf-calculus in the lqp(dc)-calculus preserves and reflects log consistency.

Theorem 2 (Equivalence of Log Consistency) *Assume T is closed. $\vdash T$ if and only if $\vdash \mathcal{T}[\![T]\!]$.*

Proof: By Lemma 6.2, we have $\vdash T$ if and only if $\vdash \mathcal{E}[\![T]\!]$, and $\vdash \mathcal{T}[\![T]\!]$ if and only if $\vdash \mathcal{E}[\![\mathcal{T}[\![T]\!]]\!]$. We have $\mathcal{E}[\![T]\!] \equiv \mathsf{new}\ \bar{t};\ \mathsf{new}\ \bar{n};\ \overline{\prod t\langle\!\langle L\rangle\!\rangle}$ for some $\{\bar{t}\}$, $\{\bar{n}\}$ and $\{\bar{L}\}$, and $\mathcal{E}[\![\mathcal{T}[\![T]\!]]\!] \equiv \mathsf{new}\ \bar{t};\ \mathsf{new}\ \bar{n};\ \overline{\prod \mathcal{T}[\![t\langle\!\langle L\rangle\!\rangle]\!]\eta}$ for some $\{\bar{t}\}$, $\{\bar{n}\}$ and $\{\bar{L}\}$, and η constructed based on the logs in $\prod t\langle\!\langle L\rangle\!\rangle$. We show that a derivation of a consistency judgement for a log entry in one calculus can be used to construct a derivation of a consistency judgement for the corresponding log entry in the other calculus. This involves induction on derivability of log entries from the context, for log entries that are required to be present by log consistency rules, and derivation of contradictions for log entries that are required to be absent. The interesting cases are for the prepare, commit and abort log entries in the atf-calculus:

Case $L \equiv L_0 \wedge \mathsf{prepare}$ **and** $L' \equiv L_0' \wedge \mathit{PreClosed} \wedge \mathit{PreCommitted}$**:** Then (CONS ATF PREP SIMPLE) is derivable in the atf-calculus if and only if (CONS PRECMT SIMPLE) and (CONS PRECLOSED) are derivable in the lqp(dc)-calculus. And (CONS ATF PREP PREDABT) is derivable in the atf-calculus if and only if (CONS PRECMT PREDABT) and (CONS PRECLOSED) are derivable in the lqp(dc)-calculus.

Case $L \equiv L_0 \wedge \mathsf{commit}$ **and** $L' \equiv L_0' \wedge \mathit{Closed}(S) \wedge \mathit{Committed}$**:** Then (CONS ATF CLOSED) is derivable in the atf-calculus if and only if (CONS CLOSED) and (CONS COMMITTED) are derivable in the lqp(dc)-calculus. The key part in the verification is that the set of all causal predecessors S computed during the translation agrees with the set of causal predecessors of t as defined by the causality relation $t' \Rightarrow t$. Since the definition of S is well-defined in the translation, the logs of all causal predecessors of t are in the context. The compatibility of the environment, $\mathbb{E}_{\mathrm{atf}}[T] \models \eta$, ensures that $\eta(t')$ maps to the immediate causal predecessors of t', according to the logs, for every predecessor of t. Therefore the definition of S in the translation agrees with the requirement of it in the (CONS CLOSED) consistency rule in the lqp(dc)-calculus: that S be the transitive closure under predecessor of the set containing t. This is enough to then show that (CONS COMMITTED) holds.

Case $L \equiv L_0 \wedge \mathsf{abort}$ **and** $L' \equiv L_0' \wedge \mathit{Aborted}$**:** Then (CONS ATF ABORT SIMPLE) is derivable in the atf-calculus if and only if (CONS ABORT SIMPLE) is derivable in the lqp(dc)-calculus. And (CONS ATF ABORT PREDABT) is derivable in the atf-calculus if and only if (CONS ABORT PREDABT) is derivable in the lqp(dc)-calculus.

□

We are now ready to verify that an encoding of a atf-calculus network in the lqp(dc)-calculus can simulate the behaviour of the former. Define the following barb predicates, where evaluation contexts $\mathbb{E}[\,\cdot\,]$ for the atf-calculus are defined in Rule (RED ATF CONG) in Fig. 11:

$$T \downarrow_n \text{ iff } T \equiv \mathbb{E}[t\langle \mathsf{send}\ n!\langle \overline{v}\rangle\rangle] \text{ where } \mathbb{E}[\,\cdot\,] \text{ does not bind } n$$
$$T \Downarrow_n \text{ iff } \exists T'.\ T \xrightarrow{*} T' \text{ and } T' \downarrow_n$$

Analogous definitions can be given for $C \downarrow_n$ and $C \Downarrow_n$.

Definition 6.3 (Compatible Environment) *Given T does not bind any transaction names. Say that the environment η is compatible with the network T, written $T \models \eta$, if the following hold:*

- *$dom(\eta)$ contains at least all of the free transaction names in $fn(T)$, and*
- *for all $t \in dom(\eta)$, if $T \equiv (\mathsf{new}\ \overline{n};\ (T' \mid t\langle\!\langle L \rangle\!\rangle))$, so the log for t is visible in T, then $t' \in \eta(t)$ if and only if $L \equiv L' \wedge t' \rightarrow t$, i.e., if and only if t' is listed as an immediate predecessor in the log for t.*

In other words, the mapping η from transaction names to sets of immediate successors is compatible with the logs in T.

Definition 6.4 (Translation of Evaluation Contexts) *Given an atf-calculus evaluation context $\mathbb{E}_{\mathrm{atf}}[\,\cdot\,]$ and a transaction T, neither of which bind transaction names. Given mapping η and a set of names $\overline{n}$, such that $dom(\eta) \cap \{\overline{n}\} = \{\}$ and $\mathbb{E}_{\mathrm{atf}}[T] \models \eta$. Define the translation of this context with respect to T, denoted $\mathcal{T}[\![\mathbb{E}_{\mathrm{atf}}[\,\cdot\,]]\!]\eta$, as follows:*

- *If $\mathbb{E}_{\mathrm{atf}}[\,\cdot\,] = [\,\cdot\,]$, then*
$$\mathcal{T}[\![\mathbb{E}_{\mathrm{atf}}[\,\cdot\,]]\!]\eta = [\,\cdot\,].$$
- *If $\mathbb{E}_{\mathrm{atf}}[\,\cdot\,] = (\mathbb{E}_0[\,\cdot\,] \mid T')$, let $\mathbb{E}_1[\,\cdot\,] = \mathcal{T}[\![\mathbb{E}_0[\,\cdot\,]]\!]\eta$ and $C' = \mathcal{T}[\![T']\!]\eta$, and define*
$$\mathcal{T}[\![\mathbb{E}_{\mathrm{atf}}[\,\cdot\,]]\!]\eta = (\mathbb{E}_1[\,\cdot\,] \mid C').$$
- *If $\mathbb{E}_{\mathrm{atf}}[\,\cdot\,] = (\mathsf{new}\ n;\ \mathbb{E}_0[\,\cdot\,])$, where n is not a transaction name, then let $\mathbb{E}_1[\,\cdot\,] = \mathcal{T}[\![\mathbb{E}_0[\,\cdot\,]]\!]\eta$, and define*
$$\mathcal{T}[\![\mathbb{E}_{\mathrm{atf}}[\,\cdot\,]]\!]\eta = (\mathsf{new}\ n;\ \mathbb{E}_1[\,\cdot\,])$$

Lemma 6.3 *Given an atf-calculus evaluation context $\mathbb{E}_{\mathrm{atf}}[\,\cdot\,]$, T and η as in the previous definition. If*
$$\mathcal{T}[\![\mathbb{E}_{\mathrm{atf}}[\,\cdot\,]]\!]\eta = \mathbb{E}_{\mathrm{lqp}}[\,\cdot\,]$$

$$\mathcal{T}[\![T]\!]\eta = C,$$

then $\mathcal{T}[\![\mathbb{E}_{\mathrm{atf}}[T]]\!]\eta = \mathbb{E}_{\mathrm{lqp}}[C]$.

Definition 6.5 (Simulation Relation) *Given an environment* η *mapping transaction names to sets of transaction names (sets of immediate causal predecessors). Define the following simulation relation between networks in the atf-calculus and networks in the lqp(dc)-calculus, where the former does not bind transaction names. The relation* $T \preceq_\eta C$ *is the largest binary relation* $\mathcal{R}$ *satisfying these conditions:*

(1) (May-testing) For $(T,C) \in \mathcal{R}$*, if* $T \downarrow_n$ *then* $C \Downarrow_n$ *for any* n*.*
(2) (Simulation) For $(T,C) \in \mathcal{R}$*, if* $T \longrightarrow T'$ *then* $C \stackrel{*}{\longrightarrow} C'$ *for some* C' *such that* $(T',C') \in \mathcal{R}$*.*
(3) (Log Equivalence) For $(T,C) \in \mathcal{R}$*, we have* $\mathcal{E}[\![\mathcal{T}[\![T]\!]\eta]\!] \equiv \mathcal{E}[\![C]\!]$*. So the two processes have isomorphic structures (up to structural equivalence) after erasing processes.*
(4) (Congruence) $\mathcal{R}$ *is a congruence: For* $(T,C) \in \mathcal{R}$*, and for any evaluation context* $\mathbb{E}_{\mathrm{atf}}[\,\cdot\,]$ *(not binding transaction names) such that* $\mathbb{E}_{\mathrm{atf}}[T] \models \eta$*, let*

$$\mathbb{E}_{\mathrm{lqp}}[\,\cdot\,] = \mathcal{T}[\![\mathbb{E}_{\mathrm{atf}}[\,\cdot\,]]\!]\eta.$$

Then $(\mathbb{E}_{\mathrm{atf}}[T], \mathbb{E}_{\mathrm{lqp}}[C]) \in \mathcal{R}$*.*

Theorem 3 (Simulation) *Given* T *not binding transaction names, and* η *compatible with it,* $T \models \eta$*. Then* $T \preceq_\eta \mathcal{T}[\![T]\!]\eta$*.*

Proof: We define the binary relation

$$\mathcal{R} = \{(T, \mathcal{T}[\![T]\!]\eta) \mid T \models \eta\}$$

and show that $\mathcal{R} \subseteq (\preceq_\eta)$, by induction on T. Congruence and log equivalence are easy consequences of the definition of $\mathcal{R}$. We concentrate on verifying simulation, by cases based on the atf-calculus reduction rule applied:

Case (RED ATF CONG)**:** By induction on the height of the evaluation context.
Case (RED ATF RECEIVE)**:** By Theorem 2, the logs are consistent in T only if they are consistent in its encoding $\mathcal{T}[\![T]\!]\eta$. So the logif in the encoding of receive chooses the else part if the log check in (RED ATF RECEIVE) succeeds in determining that the transaction is not prepared. So the encoding will execute the corresponding receive, and then apply the coinduction hypothesis to the continuation.
Case (RED ATF RECVCOMM)**:** As before, if the reduction (RED ATF RECVCOMM) is enabled, then the log check in the encoding succeeds. The message receipt in T is simulated in the encoding by unfolding the loop, receiving the message that was sent by a committed process and then confirming that it is committed. Then apply the coinduction hypothesis to the continuation.

Case (RED ATF UNDO)**:** Again log equivalence ensures that the corresponding computation in the encoding is enabled. Apply the coinduction hypothesis to the result of the reduction.

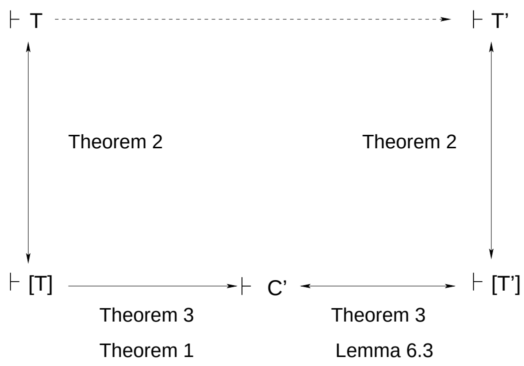
The remaining cases are similar. □

We have verified one half of a traditional operational correspondence result. We can use the preservation and reflection of log equivalence (Theorem 2), and a very simple part of the simulation result, to verify the preservation of log consistency by computation in the atf-calculus, based on the earlier result of this form for the lqp(dc)-calculus (Theorem 1).

Theorem 4 (Preservation of Log Consistency) *For any closed atf-calculus network T, if $\vdash T$ and $T \longrightarrow T'$, then $\vdash T'$.*

Proof: Suppose $\vdash T$, then Theorem 2 tells us that T and its encoding are in agreement on log consistency: $\vdash T$ if and only if $\vdash \mathcal{T}[\![T]\!]$. By the top-level restriction on transactions, we know $T = \mathsf{new}\ \bar{t};\ T_0$ for some T_0 that does not bind transaction names. Let η be an environment compatible with T_0. By Theorem 3 we know that computation in T_0 can be simulated by its encoding: $T_0 \preceq_\eta \mathcal{T}[\![T_0]\!]\eta$. So if $T_0 \longrightarrow T_0'$ for some T_0', then $\mathcal{T}[\![T_0]\!]\eta \xrightarrow{*} C_0'$ for some C' such that $T_0' \preceq_\eta C_0'$. In particular we have $\mathcal{E}[\![\mathcal{T}[\![T_0']\!]\eta]\!] \equiv \mathcal{E}[\![C_0']\!]$ by the definition of $T_0' \preceq_\eta C_0'$. Let $T' = \mathsf{new}\ \bar{n};\ T_0'$ and $C' = \mathsf{new}\ \bar{n};\ C_0'$. Therefore by Lemma 6.2 we have that $\vdash \mathcal{T}[\![T']\!]$ if and only if $\vdash C'$. But by preservation of log consistency for the lqp(dc)-calculus under computation, Theorem 1 in Sect. 5, we have that $\vdash C'$ if $\vdash \mathcal{T}[\![T]\!]$, since $\mathcal{T}[\![T]\!] \xrightarrow{*} C'$ in the lqp(dc)-calculus.

We can now putting these deductions together as illustrated in the following figure, where the arrows show the derivation of consistency:



So, we have that $\vdash T$ if and only if $\vdash \mathcal{T}[\![T]\!]$, and $\vdash \mathcal{T}[\![T]\!]$ implies $\vdash C'$, and we have $\vdash C'$ if and only if $\vdash \mathcal{T}[\![T']\!]$, and finally we have $\vdash \mathcal{T}[\![T']\!]$ if and only if $\vdash T'$ (by a second use of Theorem 2). Therefore we may conclude that $\vdash T'$. □

$L \in \text{Log}$	$::=$	$c_1 \to c_2 \mid c_1 \dashrightarrow c_2$	c_1 immediately precedes c_2
	$\mid$	$\mathit{PreClosed} \mid \mathit{Closed}(S)$	No further causal preds
	$\mid$	$\mathit{PreCommitted}$	Commitment protocol
	$\mid$	$\mathit{Committed} \mid \mathit{Aborted}$	Committed, aborted
	$\mid$	$\mathit{Undoable} \mid \mathit{Permanent}$	Anticommitment
	$\mid$	$\mathit{UndoAdmin}(S,n) \mid \mathit{UndoAuth}(n)$	Anticommitment admin
	$\mid$	$\mathit{UndoPrep}(c_1,n)$	Prepared to undo commit
	$\mid$	$\neg L$	Undone log entry
	$\mid$	$\mathit{IPreds}(S) \mid \mathit{ISuccs}(S,c,n)$	Derived predicates
$S \in \text{Set}$	$::=$	$\{v_1,\ldots,v_k\}$	

Fig. 15. Log Entry Types for the lqp(dcu)-calculus

$$
\begin{aligned}
\mathsf{Undoing}(C,c,n) &= \exists c',S.\ \exists L \in \{\mathit{UndoPrep}(c',n), \mathit{UndoAdmin}(S,n)\}.\\
&\qquad C \models c\{\!\{L\}\!\} \text{ and } C \not\models c\{\!\{\neg L\}\!\}\\
\mathsf{Undoing}(C,c) &= \exists n.\ \mathsf{Undoing}(C,c,n)\\
\mathsf{notUndoing}(C,c) &= \forall c',S,n.\ \forall L \in \{\mathit{UndoPrep}(c',n), \mathit{UndoAdmin}(S,n)\}.\\
&\qquad (C \models c\{\!\{L\}\!\} \text{ implies } C \models c\{\!\{\neg L\}\!\})\\
\mathsf{uniqUndoing}(C,c,L) &= (C \models c\{\!\{L\}\!\} \text{ and } C \not\models c\{\!\{\neg L\}\!\} \text{ and } \forall c',S,n.\\
&\qquad \forall L' \in \{\mathit{UndoPrep}(c',n), \mathit{UndoAdmin}(S,n)\}.\\
&\qquad\quad ((C \models c\{\!\{L'\}\!\} \text{ and } L \neq L') \text{ implies } C \models c\{\!\{\neg L'\}\!\}))\\
\mathsf{jointUndoing}(C,c,c') &= \exists c'',S,n.\\
&\qquad \exists L,L' \in \{\mathit{UndoPrep}(c'',n), \mathit{UndoAdmin}(S,n)\}.\\
&\qquad (C \models c\{\!\{L\}\!\},\ C \not\models c\{\!\{\neg L\}\!\} \text{ and } C \models c'\{\!\{L'\}\!\},\ C \not\models c'\{\!\{\neg L'\}\!\})\\
\mathsf{Undone}(C,c) &= (C \models c\{\!\{\mathit{Aborted}\}\!\} \text{ and } \mathsf{Undoing}(C,c))
\end{aligned}
$$

Fig. 16. Meta-predicates for Undoability

7 Anticommitment

In the lqp(dc)-calculus, a committed conclave can never abort. In this section we describe the lqp(dcu)-calculus, an extension of the lqp(dc)-calculus that provides mechanisms for *anticommitment*. A problem with transactions is their unsuitability

$$\frac{C \not\models c\{\!\{PreClosed\}\!\} \quad C \models c'\{\!\{Permanent\}\!\}}{C, c \models (c') \xrightarrow{CausalPredP} (c' \to c)} \text{(RED CAUSAL PRED PERM)}$$

$$\frac{C \not\models c\{\!\{PreClosed\}\!\} \quad C \models c'\{\!\{Undoable\}\!\}}{C, c \models (c') \xrightarrow{CausalPredT} (c' \dashrightarrow c)} \text{(RED CAUSAL PRED TENT)}$$

$$\frac{\mathsf{notUndoing}(C, c) \quad C \models c'\{\!\{c \dashrightarrow c'\}\!\}}{C, c \models (c') \xrightarrow{CausalSucc} (c \to c')} \text{(RED CAUSAL SUCC)}$$

$$\frac{C \not\models c\{\!\{PreClosed\}\!\} \quad C \models c'\{\!\{c' \to c\}\!\}}{C, c \models (c') \xrightarrow{CausalSucc} (c' \to c)} \text{(RED CAUSAL PRED CONF)}$$

$$\frac{\begin{array}{c} C \models c\{\!\{Permanent\}\!\} \text{ or } C \models c\{\!\{Undoable\}\!\} \\ S = \{c' \mid C \models c\{\!\{c' \to c\}\!\}\} \quad \{c'' \mid C \models c\{\!\{c'' \dashrightarrow c\}\!\}\} \subseteq S \end{array}}{C, c \models () \xrightarrow{PreClosed} PreClosed} \text{(RED PRECLOSED)}$$

$$\frac{S = \bigcap\{S' \mid c \in S' \text{ and } (\forall c' \in S'. \exists S''.\, C \models c'\{\!\{IPreds(S'')\}\!\} \text{ and } S'' \subseteq S')\}}{C, c \models () \xrightarrow{Closed} Closed(S)} \text{(RED CLOSED)}$$

(a) Log Append Rules

$$\frac{C \models c\{\!\{PreClosed\}\!\} \quad S = \{c' \mid C \models c\{\!\{c' \to c\}\!\}\}}{C \models c\{\!\{IPreds(S)\}\!\}} \text{(PRED PREDS)}$$

$$\frac{C \models c\{\!\{UndoPrep(c_0, n)\}\!\} \quad S = \{c' \mid C \models c\{\!\{c \to c'\}\!\}\} \quad \forall c' \in S.\, C \models c'\{\!\{c \to c'\}\!\}}{C \models c\{\!\{ISuccs(S, c_0, n)\}\!\}} \text{(PRED SUCCS)}$$

(b) Log query rules

Fig. 17. Semantics of Anticommitment in the lqp(dcu)-calculus: New Causality Rules

for long-lived applications [15], since they retain locks on database variables until the transaction eventually commits. One solution to this problem is to optimistically commit, making effects visible, and then provide a mechanism for subsequently undoing the commitment if necessary. We provide the latter mechanism through support for atomic anticommitment, atomically transforming a collection of committed conclaves to aborted conclaves. This constitutes the support for *optimistic computation* in our calculus.

We call the calculus with support for undoing commits the lqp(dcu)-calculus; it extends the lqp(dc)-calculus of dependencies and commitment with undoability of commitment. Its log entry types are provided in Fig. 15. There is one derived log entry type:

$c\{\!\{ISuccs(S,c,n)\}\!\}$ denotes that c is in a run of the anticommitment protocol uniquely identified by n, and S is the set of its immediate successors.

Commitment allows a collection of mutually dependent conclaves to commit, provided the result is causally consistent: there is no committed conclave that has aborted causal predecessors. Anticommitment allows a collection of committed conclaves to abort, provided again that the result is causally consistent: there is no aborted conclave that has committed causal successors.

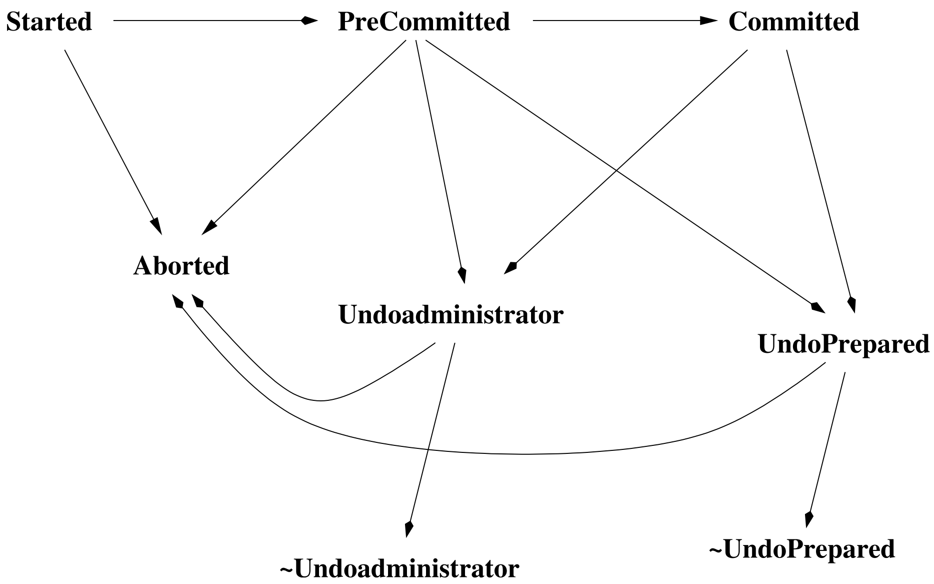
In this calculus we therefore distinguish between conclaves whose commitment can be undone, and those where this is not true. They are distinguished by log entries: conclaves of the former form have a log entry *Undoable*, while conclaves of the latter form have a log entry *Permanent*. Since "permanent" conclaves will never undo their commitment, they do not need to store their successors as the "undoable" conclaves have to.

With commitment, conclaves make a transition to the precommitted state. From there they can only make a transition to the aborted or committed state, in the former case only if a causal predecessor has aborted. It is safe for a conclave to make a transition to the committed state once all of its causal predecessors have either committed or precommitted, since the precommitted predecessors must then eventually commit. In contrast, if we are to allow anticommitment to fail (i.e., when it is not possible to abort a collection of conclaves), it is possible for a conclave that has prepared to anticommit and abort to revert back to the committed state. This leads to a potential race condition with other conclaves that have chosen to anticommit based on the readiness of this conclave to anticommit.

We give a formulation of anticommitment that avoids the aforesaid race condition. Anticommitment is based on a two-phase commit protocol, where a collection of conclaves that desire to anticommit choose (using some application-level protocol) some conclave to be the administrator for the protocol. Once the participants have entered the $UndoPrep(c,n)$ state, where c is the name of the administrator conclave and n identifies the run of the anticommitment protocol, they cannot leave this state until the administrator has make a transition to the aborted state. On the other hand, the administrator cannot make a transition to the aborted state until all participants have made a transition to the $UndoPrep(c,n)$ state.

If the anticommitment protocol fails, the administrator and participants should be able to retreat back to the committed states that they were in before they attempted anticommitment. Then for example they can retry anticommitment, perhaps under different circumstances. Since we can only add log entries, and never remove them, we model this retreat to a previous state using *negative literals* for log propositions. This is also the reason that we allow the log append operation to generate new names. The idea is that a log entry that can be undone has a unique name associated with it, and the log entry is then undone by adding the negation of that log entry to stable storage.

So the possible states that a conclave can be in are provided by the following:



The log append rules and consistency rules make use of the meta-predicates in Fig. 16. The $\mathsf{Undoing}(C,c,n)$ predicate checks that the conclave c is currently undoing its commitment, either as a participant or as an administrator in the anticommitment protocol labelled by the unique identifier n. The $\mathsf{Undoing}(C,c)$ predicate checks this condition for any n. The $\mathsf{notUndoing}(C,c)$ predicate, on the other hand, checks that any log entries flagging c as the administrator or the participant in the anticommitment protocol have been negated. The $\mathsf{uniqUndoing}(C,c,L)$ predicate checks that the log entry L is the only one, for c, for a current run of the anticommitment protocol. All other such log entries have been negated. The $\mathsf{jointUndoing}(C,c,c')$ predicate checks that both c and c' are involved in the same run of the anticommitment protocol. It should be noted that the $\mathsf{jointUndoing}(C,c,c')$ predicate is only ever used by consistence rules, never the semantics. So, it may check for the absence of a remote log without risking a race condition. Finally the $\mathsf{Undone}(C,c)$ predicate checks that c is aborted due to a run of the anticommitment protocol.

The causal dependency rules of the lqp(dc)-calculus are changed by anticommitment in the lqp(dcu)-calculus, as shown in Fig. 17. A conclave that anticommits should ensure that its (immediate) successors have aborted or will abort, in order to maintain some notion of causal consistency in the presence of the undoing of commitment. So in this system, conclaves may also have log entries of the form $c'\{\!\{c' \to c\}\!\}$, recording their successors. To ensure mutual consistency we require that a log entry recording a causal successor be justified by the presence of a log entry recording a causal predecessor. This is enforced by the antecedent of the (RED CAUSAL SUCC) rule, for adding a log entry recording a causal successor.

$$\frac{C \not\models c\{\!\{PreClosed\}\!\} \quad C \not\models c\{\!\{Permanent\}\!\}}{C, c \models () \xrightarrow{AtStUndo} Undoable} \text{(RED AT STUNDO)}$$

$$\frac{C \not\models c\{\!\{PreClosed\}\!\} \quad C \not\models c\{\!\{Undoable\}\!\}}{C, c \models () \xrightarrow{AtStPerm} Permanent} \text{(RED AT STPERM)}$$

$$\frac{C \not\models c\{\!\{PreCommitted\}\!\}}{C, c \models () \xrightarrow{AtStAbort} Aborted} \text{(RED AT STABORT)}$$

$$\frac{C \not\models c\{\!\{Aborted\}\!\}}{C, c \models () \xrightarrow{AtStPreCommit} PreCommitted} \text{(RED AT STPRECMT)}$$

$$\frac{C \models c' \Rightarrow c \quad C \models c'\{\!\{Aborted\}\!\} \quad \mathsf{notUndoing}(C, c)}{C, c \models () \xrightarrow{AtPcAbort} Aborted} \text{(RED AT PCABORT)}$$

$$\frac{C \models c\{\!\{Closed(S)\}\!\} \quad \forall c' \in S.\, C \models c'\{\!\{PreCommitted\}\!\} \quad \mathsf{notUndoing}(C, c)}{C, c \models () \xrightarrow{AtPcCommit} Committed} \text{(RED AT PCCOMMIT)}$$

Fig. 18. Commitment in the lqp(dcu)-calculus

Now there is a complication for causal consistency introduced by anticommitment: As we have said, if a conclave has committed, and subsequently anticommits and aborts, it must check first that its successors have aborted, or are aborting with it in the same run of the anticommitment protocol. This is the motivation for the successor links in the logs. But if then a conclave c has committed and has not yet anticommitted, but may do so in the future, then it must keep track of its immediate successors, even successors that make themselves dependent on c after it has committed. This introduces a synchronization requirement into the calculus: for a conclave c' to make itself dependent on c, it must first add a tentative predecessor link (log entry of the form $(c \dashrightarrow c')$ so the log of c' entails $c'\{\!\{c \dashrightarrow c'\}\!\}$), then wait for c to add a successor link (so the latter's log entails $c\{\!\{c \rightarrow c'\}\!\}$), and c' then makes its own predecessor link permanent (so its log entails $c'\{\!\{c \rightarrow c'\}\!\}$). All of this is described by the (RED CAUSAL PRED TENT), (RED CAUSAL SUCC) and (RED CAUSAL PRED CONF) rules in Fig. 17.

We ameliorate the burden of this synchronization by not requiring it for conclaves that are not undoable. So we add two states to the possible log entries: *Undoable* for conclaves that may anticommit and *Permanent* for conclaves that may not. For a conclave c that is "permanent," the (RED CAUSAL PRED PERM) may be used as before by another conclave c' to unconditionally make itself dependent on c. The lqp(dc)-calculus of the previous sections can be considered as a subset of the lqp(dcu)-calculus calculus where all conclaves have the *Permanent* log entry. Needless to say, a conclave cannot have both of these states as log entries.

The other change in the causality rules is in the definition of the immediate prede-

$$\frac{C \models c\{\!\{PreCommitted\}\!\} \quad C \models c\{\!\{Undoable\}\!\} \quad \mathsf{notUndoing}(C,c) \quad n \notin \{c, c_1, \ldots, c_k\}}{C, c \models (c_1, \ldots, c_k) \xrightarrow{AntiAdmin} \mathsf{new}\ n;\ UndoAdmin(\{c, c_1, \ldots, c_k\}, n)} \text{(RED ANTI ADMIN)}$$

$$\frac{C \models c\{\!\{PreCommitted\}\!\} \quad C \models c\{\!\{Undoable\}\!\} \quad \mathsf{notUndoing}(C,c) \quad C \models c_0\{\!\{UndoAdmin(S,n)\}\!\} \quad c \in S, c \neq c_0}{C, c \models (c_0) \xrightarrow{AntiPrep} UndoPrep(c_0, n)} \text{(RED ANTI PREP)}$$

$$\frac{\begin{array}{c} C \models c\{\!\{UndoAdmin(S,n)\}\!\} \quad C \not\models c\{\!\{\neg UndoAdmin(S,n)\}\!\} \\ S - \{c\} = \{\overline{c_k}\} \quad \forall i \in \{1,\ldots,k\}.\, C \models c_i\{\!\{ISuccs(S_i, c, n)\}\!\} \\ S_0 = \{c' \mid C \models c\{\!\{c \to c'\}\!\}\} \quad \forall c' \in S_0.\, C \models c'\{\!\{c \to c'\}\!\} \\ \forall c' \in (S_0 \cup S_1 \cup \cdots \cup S_k) - S.\, C \models c'\{\!\{Aborted\}\!\} \end{array}}{C, c \models () \xrightarrow{AntiAdmCommit} UndoAuth(n)} \text{(RED ANTI AUTHABORT)}$$

$$\frac{C \models c\{\!\{UndoAdmin(S,n)\}\!\} \quad C \not\models c\{\!\{\neg UndoAdmin(S,n)\}\!\} \quad C \models c\{\!\{UndoAuth(n)\}\!\}}{C, c \models () \xrightarrow{AntiPartCommit} Aborted} \text{(RED ANTI ADMABORT)}$$

$$\frac{C \models c\{\!\{UndoPrep(c_0,n)\}\!\} \quad C \not\models c\{\!\{\neg UndoPrep(c_0,n)\}\!\} \quad C \models c_0\{\!\{UndoAuth(n)\}\!\}}{C, c \models () \xrightarrow{AntiPartCommit} Aborted} \text{(RED ANTI PARTABORT)}$$

$$\frac{C \models c\{\!\{UndoAdmin(S,n)\}\!\} \quad C \not\models c\{\!\{UndoAuth(n)\}\!\}}{C, c \models () \xrightarrow{AntiAdmAbort} \neg UndoAdmin(S,n)} \text{(RED ANTI ADMCMT)}$$

$$\frac{C \models c\{\!\{UndoPrep(c_0,n)\}\!\} \quad C \models c_0\{\!\{\neg UndoAdmin(S,n)\}\!\}}{C, c \models () \xrightarrow{AntiPartCommit} \neg UndoPrep(c_0,n)} \text{(RED ANTI PARTCMT)}$$

Fig. 19. Anticommitment in the lqp(dcu)-calculus

cessors of a conclave. Recall that this is used in the reduction rule (RED CLOSED) for computing the transitive closure of the set of predecessors of a conclave, and closing that conclave to further additions of predecessors. The modification, as given by the (PRED PREDS) rule in Fig. 17, is that all tentative predecessor links must have been confirmed before the closure can be formed. We also add the definition of an analogous concept of immediate successors, used in anticommitment. For this we require that every successor link have a confirmed predecessor link in the immediate successor conclave, as given by rule (PRED SUCCS) in Fig. 17. Since this rule is used (by the administrator) in anticommitment to compute the set of immediate successors of the conclaves that are anticommitting, we require that the conclave be in the "prepared to anticommit" state.

The log append rules for the calculus with anti-commitment are given in Fig. 18 and 19. (RED AT STABORT) and (RED AT STPRECMT) are unchanged. A $\mathsf{notUndoing}(C,c)$ meta-predicate has been added to the rules for aborting and committing conclaves. This stops a transaction moving to one of these states part way through a run of the anti-commitment protocol.

The (RED AT STUNDO) and (RED AT STPERM) rules allow a conclave to decide at the outset if it will be undoable or permanent. The (RED AT PCCOMMIT) rule for committing replaces the (RED AT PCCOMMIT) rule for committing in Fig. 7 in Sect. 4. The only addition is an extra check that the current conclave is not already in the process of anticommiting.

The (RED ANTI ADMIN) rule allows a conclave to become an administrator in a run of the anticommitment protocol. For this, the conclave must be precommitted, must be undoable and must not already be in a run of the protocol. The (RED ANTI PREP) rule allows a conclave to enter the "prepared to anticommit" state, i.e., to become a participant in a run of the anticommitment protocol. The preconditions are as for the administrator, and in addition there must be an administrator already that includes this conclave in the set of conclaves it is administering for the protocol. The reason for requiring an administrator before a conclave can become a participant is that the creation of the former generates a new unique identifier for this run of the anticommitment protocol. This identifier is used to determine if a conclave has backed out of the protocol, by adding the negation of the administrator or participant log entry to its logs, and also matches up administrators and participants where conclaves may exit runs of the anticommitment protocol and enter new runs.

The heart of the new calculus with anticommitment is the (RED ANTI AUTHABORT) rule for authorizing the abortion of a collection of conclaves in a run of the anti-commitment protocol. Specifically this is the rule that allows the administrator to decide to abort, and all participants must then follow its lead. The administrator c is currently running a session of the anticommitment protocol identified by the unique name n. The administrator first uses the immediate successors predicate to check that all participants are in the "prepared to anticommit" state (with itself as acknowledged administrator, and in the same particular run of the protocol identified by n).

The administrator also computes its own immediate successors, and checks that all of them have matching confirmed predecessor links in their logs. The main condition to be checked then is that all of the successors computed by the administrator have aborted. If they are, the administrator makes a transition to the $\mathit{UndoAuth}(n)$ state, recording that the anticommitment protocol uniquely identified by n has succeeded. The administrator and each participant can then make a transition to the aborted state, using the (RED ANTI ADMABORT) and (RED ANTI PARTABORT) rules, respectively.

As long as it has not added the $UndoAuth(n)$ log entry, the administrator can always back out of the anticommitment protocol, using the (RED ANTI ADMCMT) rule that adds the negation of the administrator log entry to the log. This allows the conclave to subsequently participate in another run of the anticommitment protocol, either as an administrator or as a participant. Once the administrator has backed out, its negated log entry allows any of the participants to back out, using the (RED ANTI PARTCMT) rule.

7.1 Example: Sagas

A saga [39,40] is a collection of transactions $T_1, \ldots, T_k$ that execute in sequence. If transaction T_i aborts, for $i \in \{1, \ldots, k\}$, then none of the subsequent transactions execute, and moreover a collection of "antitransactions" [41] $T_{k-1}^{-1}, \ldots, T_1^{-1}$ execute in sequence. So the end of a run of a saga is either $T_1, \ldots, T_k$ or $T_1, \ldots, T_i, T_i^{-1}, \ldots, T_1^{-1}$.

Sagas are implemented fairly obviously using anticommitment. In addition anticommitment generalizes the approach of sagas to allow arbitrary dependency graphs, not just the simple linear ordering provided with sagas.

For example, the following shows a saga containing two transactions C_1 and C_2, where C_2 follows after C_1 commits, and C_1 undoes its commit if C_2 aborts:

$$\begin{array}{ll} C_1 \equiv c_1\{\mathsf{logappend}\ \langle\rangle\ \mathsf{with}\ AtStUndo; & C_2 \equiv c_2\{\mathsf{logappend}\ \langle\rangle\ \mathsf{with}\ AtStUndo; \\ \quad\quad \mathsf{logappend}\ \langle c_2\rangle\ \mathsf{with}\ CausalSucc; & \quad\quad \mathsf{logappend}\ \langle c_1\rangle\ \mathsf{with}\ CausalPredT; \\ \quad\quad P_1 \mid \mathsf{logawait}\ c_2\{\!\{Aborted\}\!\};\ P_1'\} & \quad\quad \mathsf{logappend}\ \langle c_1\rangle\ \mathsf{with}\ CausalPredP; \\ & \quad\quad \mathsf{logawait}\ c_1\{\!\{Committed\}\!\};\ P_2\} \end{array}$$

where P_1' is the code that runs the anticommitment protocol, in this case where c_1 itself is the only conclave involved in the protocol run.

8 Correctness of Anticommitment

We now consider the correctness of the lqp(dcu)-calculus. The additional and changed log consistency rules are provided in Fig. 20,21 and 22. The (CONS CAUSAL PRED PERM) rule allows a predecessor link in a log provided that the predecessor is permanent. The (CONS CAUSAL PRED TENT) rule allows a tentative predecessor link in a log if the predecessor is undoable. A confirmed predecessor link is allowed in a log, where the predecessor is undoable, if the predecessor has a matching successor link (Rule (CONS CAUSAL PRED CONF)). The latter successor link in turn requires the tentative predecessor link in the successor's log (Rule (CONS CAUSAL SUCC)). The three (CONS CAUSAL PRED) rules here replace the (CONS CAUSAL PRED) rule in Fig. 9 in Sect. 5.

$$\frac{C \models c'\{\!\{Permanent\}\!\}}{C \vdash c\{\!\{c' \rightarrow c\}\!\}} \quad \text{(CONS CAUSAL PRED PERM)}$$

$$\frac{C \not\models c\{\!\{Permanent\}\!\}}{C \vdash c\{\!\{Undoable\}\!\}} \quad \text{(CONS UNDOABLE)}$$

$$\frac{C \models c'\{\!\{Undoable\}\!\}}{C \vdash c\{\!\{c' \dashrightarrow c\}\!\}} \quad \text{(CONS CAUSAL PRED TENT)}$$

$$\frac{C \not\models c\{\!\{Undoable\}\!\}}{C \vdash c\{\!\{Permanent\}\!\}} \quad \text{(CONS PERMANENT)}$$

$$\frac{C \models c'\{\!\{Undoable\}\!\} \quad C \models c\{\!\{c' \dashrightarrow c\}\!\}}{C \vdash c'\{\!\{c' \rightarrow c\}\!\}} \quad \text{(CONS CAUSAL SUCC)}$$

$$\frac{C \models c'\{\!\{Undoable\}\!\} \quad C \models c'\{\!\{c' \rightarrow c\}\!\}}{C \vdash c\{\!\{c' \rightarrow c\}\!\}} \quad \text{(CONS CAUSAL PRED CONF)}$$

$$\frac{\begin{array}{c} C \models c\{\!\{Permanent\}\!\} \text{ or } C \models c\{\!\{Undoable\}\!\} \\ S = \{c' \mid C \models c\{\!\{c' \rightarrow c\}\!\}\} \quad \{c'' \mid C \models c\{\!\{c'' \dashrightarrow c\}\!\}\} \subseteq S \end{array}}{C \vdash c\{\!\{PreClosed\}\!\}} \quad \text{(CONS PRECLOSED)}$$

$$\frac{\begin{array}{c} \forall c' \in (fn(C) - S).\, C \not\models c' \Rightarrow c \\ \forall c' \in S.\ (C \models c' \Rightarrow c \text{ and } C \models c'\{\!\{PreClosed\}\!\}) \end{array}}{C \vdash c\{\!\{Closed(S)\}\!\}} \quad \text{(CONS CLOSED)}$$

Fig. 20. Log Consistency Rules for lqp(dcu)-calculus

The (CONS UNDOABLE) and (CONS PERMANENT) rules merely signal that these log entries are mutually exclusive, while the *PreClosed* log entry requires one or other of the form (Rule (CONS PRECLOSED)). The latter replaces the instance of (CONS PRECLOSED) in Fig. 9.

In Sect. 5 we provided two consistency rules in the lqp(dc)-calculus for the *PreCommitted* log entry: Rule (CONS PRECMT SIMPLE) for a precommitted conclave that is not aborted, and Rule (CONS PRECMT PREDABT) for a precommitted conclave that can be aborted because one of its predecessors is aborted. In Fig. 21 we add a third case in the lqp(dcu)-calculus, for when a *PreCommitted* log entry can be consistent: the conclave is aborted due to the fact that it has been undone, i.e., anticommitted. This is given by the (CONS PRECMT UNDONE) rule.

The rule for commitment in the lqp(dc)-calculus, Rule (CONS COMMITTED) in Fig. 10 in Sect. 5, required that a committed conclave be causally closed and that all of its predecessors be precommitted and not aborted. For the lqp(dcu)-calculus, we replace this with Rule (CONS COMMITTED) in Fig. 21. This rule allows a predecessor to be aborted, but if so, then either:

(1) the commitment of the current conclave must have been undone by a run of the anticommitment protocol (one that was not backed out of); or
(2) both the current conclave and its aborted predecessor are participating in the

$$\frac{C \not\models c\{\!\{Aborted\}\!\}}{C \vdash c\{\!\{PreCommitted\}\!\}}\ \text{(CONS PRECMT SIMPLE)} \qquad \frac{\mathsf{Undone}(C,c)}{C \vdash c\{\!\{PreCommitted\}\!\}}\ \text{(CONS PRECMT UNDONE)} \qquad \frac{C \not\models c\{\!\{PreCommitted\}\!\}}{C \vdash c\{\!\{Aborted\}\!\}}\ \text{(CONS ABORTED SIMPLE)}$$

$$\frac{\begin{array}{c} C \models c\{\!\{Aborted\}\!\} \\ C \models c' \Rightarrow c \quad C \models c'\{\!\{Aborted\}\!\} \quad C \not\models c'\{\!\{PreCommitted\}\!\}) \end{array}}{C \vdash c\{\!\{PreCommitted\}\!\}}\ \text{(CONS PRECMT PREDABT)}$$

$$\frac{\begin{array}{c} C \models c\{\!\{Closed(S)\}\!\} \quad \forall c' \in S.\, C \models c'\{\!\{PreCommitted\}\!\} \\ \forall c' \in S.\, C \models c'\{\!\{Aborted\}\!\} \text{ implies } (\mathsf{Undone}(C,c) \text{ or } \mathsf{jointUndoing}(C,c,c')) \end{array}}{C \vdash c\{\!\{Committed\}\!\}}\ \text{(CONS COMMITTED)}$$

$$\frac{\begin{array}{c} C \models c\{\!\{PreCommitted\}\!\} \\ C \models c' \Rightarrow c \quad C \models c'\{\!\{Aborted\}\!\} \quad C \not\models c'\{\!\{PreCommitted\}\!\}) \end{array}}{C \vdash c\{\!\{Aborted\}\!\}}\ \text{(CONS ABORTED PREDABT)}$$

$$\frac{\begin{array}{c} \mathsf{Undoing}(C,c,n) \quad C \models c'\{\!\{UndoAuth(n)\}\!\} \\ S = \{c' \mid C \models c\{\!\{c \to c'\}\!\}\} \quad \forall c' \in S.\ (c'\{\!\{Aborted\}\!\} \text{ or } \mathsf{jointUndoing}(C,c,c')) \end{array}}{C \vdash c\{\!\{Aborted\}\!\}}\ \text{(CONS ABORTED UNDONE)}$$

$$\frac{\begin{array}{c} C \models c\{\!\{UndoAdmin(S,n)\}\!\} \quad C \not\models c\{\!\{\neg UndoAdmin(S,n)\}\!\} \\ \forall n'.\, C \models c\{\!\{UndoAuth(n')\}\!\} \text{ implies } n = n' \\ S_0 = \{c' \mid C \models c\{\!\{c \to c'\}\!\}\} \quad \forall c' \in S_0.\, C \models c'\{\!\{c \to c'\}\!\} \\ S = \{\overline{c_k}\} \quad \forall i \in \{1,\ldots,k\}.\, C \models c_i\{\!\{ISuccs(S_i,c,n)\}\!\} \\ \forall c' \in (S_0 \cup S_1 \cup \cdots \cup S_k) - S.\ c'\{\!\{Aborted\}\!\} \end{array}}{C \vdash c\{\!\{UndoAuth(n)\}\!\}}\ \text{(CONS UNDO AUTH)}$$

Fig. 21. Log Consistency Rules for lqp(dcu)-calculus (cont'd)

same run of the anticommitment protocol; since the predecessor has aborted when it was precommitted, it must have aborted because of the anticommitment protocol; so the administrator has aborted and the current conclave will also abort (if it makes progress), but it will not back out of the anticommitment protocol.

Fig. 10 in Sect. 5 provided two rules for the consistency of log entries for abortion in the lqp(dc)-calculus. The (CONS ABORTED SIMPLE) rule allowed a conclave to abort if it had not precommitted. The (CONS ABORTED PREDABT) rule allowed a precommitted conclave to be aborted if one of its predecessors was aborted. In Fig. 21 we add a third rule for the lqp(dcu)-calculus, Rule (CONS ABORTED UN-

$$\frac{\begin{array}{c}C \models c\{\!\{PreCommitted\}\!\} \quad C \models c\{\!\{Undoable\}\!\} \quad C \models c\{\!\{\neg UndoAdmin(S,n)\}\!\} \\ \forall S'.\, C \models c\{\!\{UndoAdmin(S',n)\}\!\} \text{ implies } S = S'\end{array}}{C \vdash c\{\!\{UndoAdmin(S,n)\}\!\}} \quad (\textsc{Cons UndoAdm Past})$$

$$\frac{\begin{array}{c}C \models c\{\!\{PreCommitted\}\!\} \quad C \models c\{\!\{Undoable\}\!\} \quad (\forall c' \in S.\, C \not\models c'\{\!\{\neg UndoPrep(c,n)\}\!\}) \\ \mathsf{uniqUndoing}(C, c, UndoAdmin(S,n)) \\ \forall S'.\, C \models c\{\!\{UndoAdmin(S',n)\}\!\} \text{ implies } S = S'\end{array}}{C \vdash c\{\!\{UndoAdmin(S,n)\}\!\}} \quad (\textsc{Cons UndoAdm Curr})$$

$$\frac{C \models c\{\!\{UndoAdmin(S,n)\}\!\} \quad C \not\models c\{\!\{UndoAuth(n)\}\!\}}{C \vdash c\{\!\{\neg UndoAdmin(S,n)\}\!\}} \quad (\textsc{Cons UndoAdm Neg})$$

$$\frac{\begin{array}{c}C \models c\{\!\{PreCommitted\}\!\} \quad C \models c\{\!\{Undoable\}\!\} \\ C \models c\{\!\{\neg UndoPrep(c_0,n)\}\!\} \quad C \models c_0\{\!\{\neg UndoAdmin(S,n)\}\!\} \quad c \in S, c \neq c_0 \\ \forall c_0'.\, C \models c\{\!\{UndoPrep(c_0',n)\}\!\} \text{ implies } c_0 = c_0'\end{array}}{C \vdash c\{\!\{UndoPrep(c_0,n)\}\!\}} \quad (\textsc{Cons UndoPrep Past})$$

$$\frac{\begin{array}{c}C \models c\{\!\{PreCommitted\}\!\} \quad C \models c\{\!\{Undoable\}\!\} \\ \mathsf{uniqUndoing}(C, c, UndoPrep(c_0,n)) \\ C \models c_0\{\!\{UndoAdmin(S,n)\}\!\} \quad c \in S, c \neq c_0 \\ \forall c_0''.\, C \models c\{\!\{UndoPrep(c_0'',n)\}\!\} \text{ implies } c = c_0''\end{array}}{C \vdash c\{\!\{UndoPrep(c_0,n)\}\!\}} \quad (\textsc{Cons UndoPrep Curr})$$

$$\frac{C \models c\{\!\{UndoPrep(c_0,n)\}\!\} \qquad C \models c_0\{\!\{\neg UndoAdmin(S,n)\}\!\}}{C \vdash c\{\!\{\neg UndoPrep(c_0,n)\}\!\}} \quad (\textsc{Cons UndoPrep Neg})$$

Fig. 22. Log Consistency Rules for lqp(dcu)-calculus (cont'd)

DONE), that allows a conclave to be aborted because of anticommitment. In this case, any immediate successor must also be aborted, or else be in the same run of the anticommitment protocol as the current conclave. As in the previous case of the commitment rule, since the current conclave aborted during the current run of the anticommitment protocol, the successor can only conclude the protocol by aborting.

The (CONS UNDOADM PAST) and (CONS UNDOPREP PAST) rules in Fig. 22 check the consistence of undo log entries for administrators and participants that have finished backing out of a run of the anticommmmitment protocol. The conclave must be precommitted and undoable, and in the case of a participant prepared to anticommit, there must have been an administrator for that run of the protocol that also backed out of the protocol. For a current administrator, Rule (CONS UN-

DOADM CURR), none of the participants can have backed out of the protocol if the administrator has not backed out. Also of course the current conclave cannot be involved in any other run of the anticommitment protocol. For a current participant, Rule (CONS UNDOPREP CURR), there must be an administrator for the protocol that has agreed to include this conclave as a participant.

To extend the consistency preservation result from Sect. 4, Theorem 1, to the system with anticommitment, we need to extend the proof of Lemma 5.2. The extended proof is provided in a technical report [35]. The proof of the following is similar to the proof of Theorem 1, using the proof of the aforesaid lemma for the base cases in the induction.

Theorem 5 *For the lqp(dcu)-calculus, if* $\vdash C_1$ *and* $C_1 \xrightarrow{*} C_2$*, then* $\vdash C_2$.

9 Related Work and Conclusions

As alluded to in the introduction, there is a large body of literature on various forms of transaction models, particularly for long-lived applications. See e.g. [15] for a survey. We do not claim that the lqp(dc)-calculus or the lqp(dcu)-calculus is uniformly better than other transaction models that have been proposed, and in fact there are aspects of various transaction models that are missing. For example the ACTA model [42,43], that attempts to unify many forms of extended transaction models, has both success and failure dependencies between transactions, while the lqp(dcu)-calculus has only failure dependencies. The point of the current article is rather, first, to show how various transaction models can be decomposed into building blocks from which more complicated transaction models can be built up, and second, to show how the abstraction of logs in the lqp($\cdot$)-calculus family isolates the communication requirements upon which these building blocks rely.

Numerous process algebras have been proposed as the foundations of programming languages for wide-area applications. Most of the work in the literature is based on mobile computation and mobile code to deal with latency and firewall problems [4–8]. Much of the aforesaid 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 [20–23], it has assumed a fail-stop model of failures that is not always a good match for programming in Internet environments. The M-calculus [44] gives broader control over locations and makes it possible to simulate network failure as well as a fail-stop model. The synchronous message sending operations of CCS and the pi-calculus require global atomic commitment and therefore are unimplementable in an asynchronous distributed system [17,18]. Palamidessi [28] shows that the leadership election problem can be solved in the pi-calculus, but not in the asynchronous pi-calculus. Herescu and Palamidessi [45] describe a variant of the asynchronous pi-calculus with a probabilistic choice operation, and show that it is possible to implement a leadership election algorithm in this calculus.

There are many ways in which conclaves differ from ambients, from the Ambient Calculus [4], and we only mention the two that are most relevant. First, ambients do not have the distributivity rule for parallel composition. This reflects the different objectives of the calculi: ambients want to make all communication local, whereas we do not want boundaries for atomic failure to interfere with communication. Therefore we do not complicate our calculus with operations for "navigating" conclaves, whereas such navigation is at the heart of the ambient calculus. Second, conclave execution cannot be nested within another conclave. We do not pursue this complication of the calculus because we take conclaves to represent the most basic area of distribution that could fail or succeed as a single unit. A desire for nesting might be motivated by something analogous to nested transactions [10,11]. However nested transactions are sufficiently complicated in a global computing environment that we prefer to build them up from simpler notions, as alluded to in Sect. 9.

The $D\pi$-calculus of Riely and Hennessy [23] is perhaps the closest to our language. They have a notion of locations, a rule for distributing parallel composition over locations (as in the lqp($\cdot$)-calculus). They also have an operation for moving processes between locations, which is one application of the forking construct in the lqp($\cdot$)-calculus. The most important part of their language is the ability to detect failures at remote locations; they adopt the fail-stop model for their failure semantics. This is the biggest difference between the $D\pi$-calculus and the lqp($\cdot$)-calculus: the former provides a particular failure model and relies on failure detection and message-passing to handle distributed coordination. The lqp($\cdot$)-calculus leaves the failure model unspecified, and focuses on providing a framework for defining protocols for distributed coordination.

Concurrent constraint languages [46–49] replace message buffers with a global store of constraints, with ask and tell operations for querying the store and adding constraints to the store, respectively. Our model does not replace message buffers in the asynchronous pi-calculus, and indeed we expect that eventually (as alluded to below) remote querying of logs would be implemented using message-passing. Concurrent constraint programs may make the store inconsistent; our operations for modifying stable storage are designed to preserve log consistency, as verified by Theorem 1.

Needless to say, transactions and atomic commitment can be implemented in distributed programming languages, and therefore in calculi that are intended to be "kernel languages" for distributed programming. Berger and Honda provide an implementation of two phase commit in the pi-calculus [50]. Along the way they extend the pi-calculus with extra syntax for message loss, timers and process failure. Bruni et al [51] give an implementation of a scripting language, for transactions that use two-phase commit, in the Join calculus, [5]. Both this work, and Berger and Honda's pi-calculus model, aim to make a test-bed for a basic instance of a single kind of atomic commitment protocol. Busi et al [52] propose a formal modelling of transactions in JavaSpaces based on process-calculi techniques. The focus of these efforts is different from the work presented here, which proposes a programming model and a set of abstractions for building different forms of transactions, and different atomic commitment protocols, in global computing environments.

Field and Varela [53] give a semantics for a domain-specific language for programming distributed transactions. The semantics includes a two-phase commit protocol, as well as tracking of failure dependencies and process rollback. This approach is somewhat more high level than the approach of the lqp($\cdot$)-calculus: it commits to a particular protocol for global agreement, it assumes powerful but potentially expensive run-time facilities (distributed process rollback), and it does not address the issue of decoupling protocols from their communication requirements. As such the language of Field and Varela is a potentially useful source language that could be translated to the lqp(dc)-calculus, for example, in a similar manner to the atf-calculus.

Perhaps the calculus that at least superficially is closest to ours is the join-calculus [5,21]. This calculus allows processes to reflect new process descriptions into the semantics, based on multiset rewriting rules where the multiset contains buffered messages. Related calculi include KLAIM [54], a distributed language based on the Linda primitives [55]. However the intention and therefore the mechanisms of the two approaches are quite different. Our use of a multiset of propositions to model stable storage is intended to isolate the communication requirements of fault-tolerance protocols, and the calculus has a predefined collection of rules for adding new log entries, with an emphasis on preserving the consistency of the logs. The distributed join calculus does consider primitives for fault tolerance, but they are based on the fail-stop model that only holds for synchronous distributed systems.

An interesting further direction suggested by the join calculus would be to allow applications to define new log entry types, and new rules for adding those log entries to logs during execution. There are interesting security issues with such an idea: What relationships are allowed between new log entry types and existing log entry types, and what log consistency properties could be asserted by applications? What responsibility does an application have to ensure that any log extension rules that it adds preserve log consistency? The join calculus allows new atom types to be defined, by creating new ports, and any process can add atoms (send messages to a port), although receipt of such messages is restricted to the original site. In contrast with the join calculus, new rules for adding log entries of new user-defined types would be global (available to all processes), rather than local as in the join calculus. This is an area for further work.

There are several other directions for further work. One direction is to consider how to extend this model with support for nested transactions and partial failures [10,11]. A notion of equivalence would also be useful for this calculus, particularly a recursive description analogous to the bisimulation method for CCS. Finally, Our approach isolates the remote communication aspects of conclaves to the querying of logs of remote conclaves. We are developing an approach to assigning the application the responsibility of providing the remote communication for this querying, without compromising the security of transitions that affect stable storage. We hope to have the opportunity to report on these developments in subsequent articles.

Acknowledgements: Thanks to Cedric Fournet, Andrew Gordon and Sanjiva Prasad for helpful conversations. Thanks to the anonymous reviewers for their excellent comments and suggestions.

References

[1] L. Cardelli, Abstractions for mobile computation, in: J. Vitek, C. Jensen (Eds.), Secure Internet Programming: Security Issues for Distributed and Mobile Objects, Vol. 1603 of Lecture Notes in Computer Science, Springer-Verlag, 1999.

[2] A. Birrell, G. Nelson, S. Owicki, E. Wobber, Network objects, in: Symposium on Operating Systems Principles, ACM Press, 1993, pp. 217–230.

[3] K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo, A. Wollrath, The Jini Specification, Addison-Wesley, 1999.

[4] L. Cardelli, A. Gordon, Mobile ambients, in: M. Nivat (Ed.), Foundations of Software Science and Computational Structures, Vol. 1378 of Lecture Notes in Computer Science, Springer-Verlag, 1998, pp. 140–155.

[5] C. Fournet, G. Gonthier, The reflexive chemical abstract machine and the join-calculus, in: Proceedings of the 23rd ACM Symposium on Principles of Programming Languages, ACM, St. Petersburg Beach, Florida, 1996, pp. 372–385.

[6] G. Castagna, J. Vitek, A calculus of secure mobile computations, in: Internet Programming Languages, Lecture Notes in Computer Science, Springer-Verlag, 1999.

[7] J. Riely, M. Hennessy, Trust and partial typing in open systems of mobile agents, in: Proceedings of ACM Symposium on Principles of Programming Languages, 1999.

[8] M. Hennessy, J. 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.

[9] B. Liskov, Distributed programming in Argus, Communications of the ACM 31 (3) (1988) 300–312.

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

[11] N. Lynch, M. Merritt, W. Weihl, A. Fekete, Atomic Transactions, Morgan-Kaufman, 1994.

[12] D. Detlefs, M. Herlihy, J. Wing, Inheritance of synchronization and recovery properties in avalon/c++, IEEE Computer (1988) 57–69.

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

[14] D. Krieger, R. Adler, The emergence of distributed component platforms, IEEE Computer 31 (3) (1998) 43–53.

[15] A. K. Elmagarmid (Ed.), Database Transaction Models for Advanced Applications, Morgan Kaufmann, 1992.

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

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

[18] V. Hadzilacos, On the relationship between the atomic commitment and consensus problems, in: B. Simons, A. Z. Spector (Eds.), Fault-Tolerant Distributed Computing, Vol. 448 of Lecture Notes in Computer Science, Springer-Verlag, 1990, pp. 201–208.

[19] T. D. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, Journal of the ACM 43 (4) (1996) 685–722.

[20] R. Amadio, S. Prasad, Localities and failures, in: P. S. Thiagarajan (Ed.), Proceedings of 14th Conference on Foundations of Software Technology and Theoretical Computer Science, no. 880 in Lecture Notes in Computer Science, Springer-Verlag, 1995, pp. 205–216.

[21] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, D. Rémy, A calculus of mobile agents, in: 7th International Conference on Concurrency Theory (CONCUR'96), Springer-Verlag, Pisa, Italy, 1996, pp. 406–421, lNCS 1119.

[22] R. M. Amadio, An asynchronous model of locality, failure and process mobility, in: COORDINATION'97, Vol. 1282 of Lecture Notes in Computer Science, Springer-Verlag, 1997.

[23] J. Riely, M. Hennessy, Distributed processes and location failures, in: Proceedings of the International Conference on Automata, Languages and Programming, 1997.

[24] R. Milner, The polyadic π-calculus: A tutorial, in: F. L. Bauer, W. Brauer, H. Schwichtenberg (Eds.), Logic and Algebra of Specification, Vol. 94 of Computer and Systems Sciences, Springer-Verlag, 1993, pp. 203–246.

[25] K. Honda, M. Tokoro, An object calculus for asynchronous communication, in: European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, Springer-Verlag, 1991, pp. 133–147.

[26] D. Sangiorgi, Asynchronous process calculi: The first-order and higher-order paradigms, Theoretical Computer Science (253).

[27] T. Chothia, I. Stark, A distributed calculus with local areas of communication, in: High Level Concurrent Languages, 2000.

[28] C. Palamidessi, Comparing the expressive power of the synchronous and the asynchronous pi-calculus, in: Proceedings of ACM Symposium on Principles of Programming Languages, ACM Press, 1997.

[29] S. B. Davidson, Optimism and consistency in partitioned database systems, ACM Trans. Database Syst. 9 (3) (1984) 456–481.

[30] R. Schwarz, F. Mattern, Detecting causal relationships in distributed computations: In search of the Holy Grail, Tech. Rep. SFB124-15/92, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany (1992).

[31] L. Lamport, Time, clocks and the ordering of events in a distributed system, Communications of the ACM 21 (7) (1978) 558–565.

[32] R. van Renesse, Causal controversy at Le Mont St.-Michel, Operating Systems Review 27 (2) (1993) 44–53.

[33] D. Cheriton, D. Skeen, Understanding the limitations of causally and totally ordered communication, in: Symposium on Operating Systems Principles, 1993.

[34] C. Pu, G. Kaiser, N. Hutchinson, Split-transactions for open-ended activities, in: VLDB Conference, 1988, pp. 26–37.

[35] T. Chothia, D. Duggan, Abstractions for fault-tolerant global computing, Tech. Rep. CS Report 2003-3, Stevens Institute of Technology (2003).

[36] D. Duggan, Atomic failure in wide-area computation, in: Formal Methods in Open Object-Based Distributed Systems (FMOODS), Kluwer, Stanford, California, 2000.

[37] J. Stamos, F. Cristian, A low-cost atomic commit protocol, in: IEEE Symposium on Reliable Distributed Systems, 1990.

[38] U. Nestmann, B. C. Pierce, Decoding choice encodings, in: U. Montanari, V. Sassone (Eds.), CONCUR '96: Concurrency Theory, 7th International Conference, Vol. 1119, Springer-Verlag, Pisa, Italy, 1996, pp. 179–194.

[39] H. Garcia-Molina, K. Salem, Sagas, in: ACM SIGMOD International Conference on Management of Data, 1987, pp. 249–259.

[40] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, K. Salem, Modeling long-running activities as nested sagas, Bulletin of the IEEE Technical Committee on Data Engineering 14 (1) (1991) 14–18.

[41] H. Korth, E. Levy, A. Silberschatz, Compensating transactions: A new recovery paradigm, in: VLDB Conference, 1990, pp. 95–106.

[42] P. K. Chrysanthis, K. Ramamritham, ACTA: a framework for specifying and reasoning about transaction structure and behavior, in: Proceedings of ACM SIGMOD, 1990, pp. 194–203.

[43] P. K. Chrysanthis, K. Ramamritham, Synthesis of extended transaction models using ACTA, ACM Transactions on Database Systems 19 (3) (1994) 450–491.

[44] A. Schmitt, J.-B. Stefani, The M-calculus: A Higher-Order Distributed Process Calculus, in: Proceedings of ACM Symposium on Principles of Programming Languages, 2003.

[45] O. M. Herescu, C. Palamidessi, Probabilistic asynchronous π-calculus, in: J. Tiuryn (Ed.), Proceedings of FOSSACS 2000 (Part of ETAPS 2000), Lecture Notes in Computer Science, Springer-Verlag, 2000, pp. 146–160.

[46] V. Saraswat, M. Rinard, Concurrent constraint programming, in: Proceedings of ACM Symposium on Principles of Programming Languages, 1990.

[47] V. Saraswat, M. Rinard, P. Panangaden, Semantic foundations of concurrent constraint programming, in: Proceedings of ACM Symposium on Principles of Programming Languages, 1991.

[48] F. Bueno, M. V. Hermenegildo, U. Montanari, F. Rossi, Partial order and contextual net semantics for atomic and locally atomic cc programs, Science of Computer Programming 30 (1998) 51–82.

[49] F. de Boer, M. Gabbrielli, E. Marchiori, C. Palamidessi, Proving concurrent constraint programs correct, ACM Transactions on Programming Languages and Systems 19 (1998) 685–725.

[50] M. Berger, K. Honda, The two-phase commitment protocol in an extended pi-calculus, in: Proceedings of EXPRESS '00: Expressiveness in Concurrency, Electronic Notes in Theoretical Computer Science, Elsevier, pp. 105–130.

[51] R. Bruni, C. Laneve, U. Montanari, Orchestrating transactions in the Join calculus, in: CONCUR 2002, 13th International Conference on Concurrency Theory, Lecture Notes in Computer Science, Springer-Verlag, 2002.

[52] N. Busi, R. Gorrieri, G. Zavattaro, On the serializability of transactions in JavaSpaces, in: ConCoord 2001, International Workshop on Concurrency and Coordination, Vol. 54 of ENTCS, 2001.

[53] J. Field, C. Varela, Towards a programming model for building reliable systems with distributed state, in: Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA), 2002.

[54] R. D. Nicola, G. Ferrari, R. Pugliese, KLAIM: A kernel language for agents interaction and mobility, IEEE Transactions on Software Engineering 24 (5) (1998) 315–330.

[55] D. Gelernter, Generative communication in Linda, ACM Transactions on Programming Languages and Systems 7 (1) (1985) 80–112.