Transactors: A Programming Model for Maintaining Globally Consistent Distributed State in Unreliable Environments

John Field
IBM T.J. Watson Research Center
jfield@watson.ibm.com

Carlos A. Varela
Department of Computer Science
Rensselaer Polytechnic Institute
cvarela@cs.rpi.edu

ABSTRACT

We introduce *transactors*, a fault-tolerant programming model for composing loosely-coupled distributed components running in an unreliable environment such as the internet into systems that reliably maintain globally consistent distributed state. The transactor model incorporates certain elements of traditional transaction processing, but allows these elements to be composed in different ways without the need for central coordination, thus facilitating the study of distributed fault-tolerance from a semantic point of view. We formalize our approach via the τ-calculus, an extended lambda-calculus based on the *actor* model, and illustrate its usage through a number of examples. The τ-calculus incorporates constructs which distributed processes can use to create globally-consistent *checkpoints*. We provide an operational semantics for the τ-calculus, and formalize the following safety and liveness properties: first, we show that globally-consistent checkpoints have equivalent execution traces without any node failures or application-level failures, and second, we show that it is possible to reach globally-consistent checkpoints provided that there is some bounded failure-free interval during which checkpointing can occur.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*operational semantics, process models*; D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart, fault-tolerance*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*

General Terms

Languages, Reliability, Design, Theory

Keywords

Distributed state, Transactor, Tau-calculus, Actor

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Extended version of POPL'05 proceedings paper, January 12–14, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-58113-830-X/05/0001 ...$5.00.

1. MOTIVATION

Many distributed systems must maintain *distributed state*. By this, we mean that the states of several distributed components in a network-connected system are interdependent on one another. The classical example of such a scenario is a bank transaction involving the transfer of money from one account to another, where we must ensure that it is not possible (even in the presence of a system failure) for one account to be debited without a corresponding credit being made to the other account, and vice-versa.

Ensuring that these interrelated states are maintained in a *consistent* way in a wide-area network—where transmission latencies may be high, and where node and link failures are relatively common occurrences—is difficult. By exposing key semantic concepts related to maintenance of distributed state in a common, well-founded *language*, rather than relegating these issues to system or middleware, composite distributed applications can reason about the failure semantics of their components, and, if appropriate, supply extra protocol layers (e.g., logging, rollbacks, retries, replication, etc.) to add additional reliability.

To better illustrate the complexity of maintaining distributed state in a loosely-coupled distributed system, consider a collection of web services that are combined dynamically to manage the purchase of a house. Such a purchase is a complex multi-step transaction involving many interacting participants. In the U.S., it would not be unusual for the list of participants to include, in addition to the buyer and seller, real estate agents, lawyers, banks, inspectors, mortgage brokers, mortgage issuers, municipal authorities, and more.

Today, many of the steps required to purchase a house entail tedious requests and responses for information via telephone calls, faxes and paper documents. A few of those steps, such as searching for candidate houses to purchase and choosing among various mortgage lenders, can now be facilitated by interactive online services. However, in the future, it should be possible for virtually all the information generated during the process to be exchanged and managed electronically.

There are a number of challenges to designing a distributed web services infrastructure to support complex transactions such as house purchases: We would like to allow various services to be assembled dynamically, to manage the flow of information over an extended period of time, to cope with the possibility of process or network failures while the transaction is pending, to allow for rectification of various failures of semantic consistency (e.g., an attempt to overdraw a bank account), and to ensure that all of the parties complete the transaction in a consistent, durable state.

Figures 1 and 2 depict a subset of the operations that might be

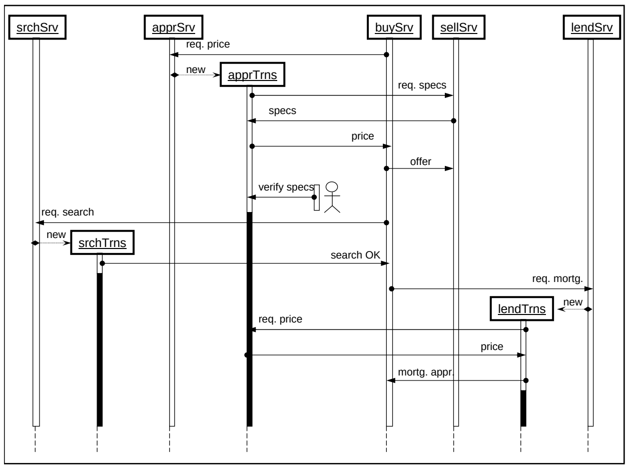

Figure 1: A collection of interacting web services managing (part of) the purchase of a house. buySrv, sellSrv, lendSrv, apprSrv, and srchSrv represent services for the buyer, seller, lender, appraisal service, and title search service, respectively. lendTrns, apprTrns, and srchTrns represent sub-processes spawned specifically to manage the interaction with buySrv. Portions of the vertical process bars that are black represent "stable states".

performed by a collection of web services involved in the negotiation of a house purchase, and serve to illustrate many of the issues that arise in building an infrastructure to support such services. We will consider such services to be concurrent processes that can send and receive messages to other processes as well as spawn new processes. Fig. 1 depicts a portion of a successful negotiation, while Fig. 2 depicts the more interesting case (from our perspective) of failure and subsequent recovery of several sub-processes. In the figures, the vertical bars labeled by buySrv and sellSrv represent web services acting on behalf of the buyer and seller, respectively. lendSrv, apprSrv, and srchSrv represent web services for a lender, appraisal service, and title search service, respectively. lendTrns, apprTrns, and srchTrns represent sub-processes spawned by the lender, appraisal service, and search service specifically to manage the interaction with the particular buyer in this example. Horizontal arrows depict messages sent between processes or the creation of new processes. Portions of the vertical process bars that are black represent "stable states", where the state maintained by the process should not subsequently change. Process rollback (arising from various forms of failure) is depicted by dashed diagonal arrows.

We will now consider the scenario of Fig. 2 in more detail. Some of the key steps in this process are as follows:

1. The buyer chooses a candidate house, and initiates the buySrv web service (perhaps via a real estate agent) to manage the house purchase process.

2. In order to determine an appropriate price to offer the seller, buySrv contacts an appraisal service, apprSrv, whose job is to estimate the "market value" of the house.

3. apprSrv spawns a process specifically to manage the interaction with buySrv. apprSrv requests basic information about the house (location, size, condition, amenities, etc.) from the seller's web service, sellSrv.

4. apprSrv combines the house specifications with historical information in a database of recent house purchases to compute a "tentative" market price, which is stored and transmitted to the seller. The tentative price is normally accurate (within the limits of subjectivity that such a valuation entails). However, before producing a "definitive" appraised value, the appraisal process requires an on-site visit (by a person) to the house to verify the accuracy of the specifications originally sent by the seller's service.

5. The buyer decides to offer the seller the price computed by the appraisal service. As is typically the case, the offer is made contingent on the buyer's ability to find a mortgage lender and on the absence of misrepresentations about the house's specifications and condition. This offer is transmitted to sellSrv.

6. buySrv connects to a title search service, srchSrv, which ensures that the owner of the house has the legal right to sell it. srchSrv spawns a subprocess, srchTrns, to manage the interaction with buySrv.

7. buySrv contacts a mortgage lender service, lendSrv, for a quote on a mortage, giving the lender the (electronic) address of apprTrns (the lender service needs professional appraisal information to ensure that the buyer is not borrowing more money than the house is worth as collateral). lendSrv spawns a subprocess, lendTrns, to manage the interaction with buySrv.

8. The appraiser visits the house in person, and discovers that the house specifications transmitted by the seller's agent were inaccurate. The new information results in a house value lower than the tentative value computed earlier. Since the appraisal service already transmitted the tentative value, which is now incorrect, it (voluntarily) *rolls back* to its initial state, then (re-) processes the human-gathered house specification information.

9. lendTrns contacts apprTrns to determine the appraised price of the house. At this point, the price information used to approve the mortgage differs from the price information originally transmitted by apprTrns to the buyer.

10. The lender approves the mortgage, and sends the approval notice to the buyer.

After step 10 above, buySrv has received two pieces of semantically *inconsistent* information: the tentative price transmitted initially to apprTrns, and the mortgage approval message, which was computed based on the new price information computed by apprTrns after rolling back. As a result of this inconsistency, buySrv must now roll back in order to attempt to re-establish a consistent state. After buySrv rolls back, the buyer may, e.g., wish to invoke the contingency clause in the contract to renegotiate the sale price.

The transactor model serves to maintain *dependence* information needed to detect semantic inconsistencies such as that depicted in Fig. 2, and to cause the rollback of the buySrv process to occur automatically. In addition to such "semantic" failures, process or network failures during the course of the transaction might cause information loss that requires an orderly re-establishment of the transaction's distributed state. Note, however, that certain steps of the transaction, such as the title search, need not be renegotiated after a semantic or system failure, since the results of the search

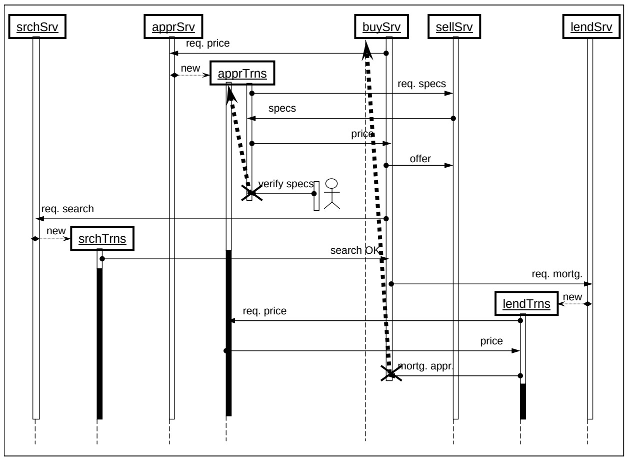

Figure 2: A house purchase scenario involving the "semantic failure" and rollback of the apprTrns sub-process due to inconsistent information about house specifications. This ultimately results in the rollback of the buySrv process itself, due to the inconsistency between the appraised price used to initiate the buying process, and the differing price incorporated in the mortgage approval sub-process.

are unaffected by the inconsistent appraisal values. Such steps can be *committed* early in the process, while other steps (such as the final transfer of the purchase price) might require mutual agreement between multiple parties to be reached before becoming durable and binding.

In this paper, we describe the *transactor* model, a fault-tolerant programming model for composing loosely-coupled distributed components running in an unreliable environment such as the internet into systems that reliably maintain consistent distributed state. Our model is *not* concerned with certain aspects of traditional "ACID" transactions [13] such as isolation or certain forms of atomicity. While such features are beyond the scope of this paper, they can be explicitly coded in our model if desired, e.g., in a manner similar to [11]; instead, we focus on ensuring consistency of distributed state in the presence of certain types of node and network failures. In particular, we assume that a node fails either by stopping, or by reverting to a programmatically-defined *checkpoint* saved to stable storage, then restarting.

The remainder of the paper is structured as follows: Section 2 introduces related work. Section 3 informally describes the transactor model. Section 4 introduces the syntax of the τ-calculus, an extended lambda-calculus based on the *actor* model. Section 5 illustrates some representative transactor examples. Section 6 provides an operational semantics for the τ-calculus. Section 7 formalizes safety and liveness properties of the model. Finally, Section 8 concludes with a discussion and potential future directions.

2. RELATED WORK

The transactor model is based on the *actor* model introduced by Hewitt [15], and further refined and developed by Agha et al. [1, 3, 22]. Actors are inherently independent, concurrent, and autonomous which enables efficiency in parallel execution [16] and facilitates mobility [2]. The actor model and languages provide a very useful framework for understanding and implementing open distributed systems. Transactors can be regarded as a *coordination model* [12, 11, 24], in the sense that they are intended primarily to express the semantics of the *interactions* among various distributed components, rather than to describe the computations local to a node in the system.

Traditionally, distributed state maintenance has been viewed primarily as a systems or "middleware" [5] problem, in which, e.g., system infrastructure for message-passing provides guaranteed message delivery on an unreliable network substrate [6, 23], or where distributed databases or transaction systems support the illusion of shared, atomically-updatable state across multiple nodes [17, 20]. A number of projects are underway to realize distributed programming models for the internet, e.g., the *web services* model based on exchange of XML data [26].

Distributed transaction management systems, e.g., those that implement the XA system-neutral transaction API [27], typically require that all of the participants in the transaction coordinate their work with a pre-designated *transaction manager*, and that every transaction has well-defined beginning and end points. These properties make it difficult to build open distributed systems where the topology of the system is determined dynamically, where the scope of—and even the need for—a transaction is situation-dependent, and where transactional and non-transactional components can easily interact. While there is much existing foundational work on languages for concurrent, and to a lesser extent, distributed systems (e.g., actors [1, 3], the π-calculus [18], the join calculus [10], and mobile ambients [7]), formalisms that provide primitives for reasoning about the consistency of distributed state in the presence of failures are not well developed. At the other end of the spectrum, distribution in "industrial" languages or language models, e.g., Java RMI [21], Jini [25], CORBA [19], and COM+, is generally based on remote procedure call models that have limited mechanisms for dealing with failure, and are best suited for tightly-coupled, centrally-managed applications.

Liskov's Argus language [17] incorporates constructs for maintenance of distributed state (via *nested transactions*). Liskov introduced two principal abstractions: guardians and actions. A guardian is an abstract object whose purpose is to encapsulate a resource or resources. Special procedures, called handlers, can be used to access a guardian. An action is essentially a nested atomic transaction. Argus essentially provides a programming interface onto centrally-managed nested transactions. By contrast, with transactors, we intend to uniformly model a variety of failure-management techniques, including transactions and applications with weaker consistency semantics. Haines et al. designed an extension to ML to modularly support first-class transactions [14]. That is, atomicity, isolation and durability properties can be composed as desired. We are concerned with distributed state consistency and durability, and do not explicitly model isolation. Atomicity within a transactor is inherited from the actor model, where each transactor represents a unit of concurrency and processes only one message at a time. Other actor-based abstractions (such as synchronizers [11]) can be used to provide atomicity for actions performed by groups of co-related actors.

Chotia and Duggan's abstractions for fault-tolerant global computing [8] include *conclaves* as groups of correlated processes which fail atomically, and *logs* which abstract over persistent storage. Berger and Honda provide an extension to the π-calculus to model the two-phase commitment protocol [4]. While the motivation of their work is similar to ours, the approaches are quite different. The transactor model does not assume atomicity in process group failures: transactors can fail independently and causal dependencies are carried along with messages to ensure that only globally consistent checkpoints can be reached by application-level protocols. Our calculus also enables reasoning about and composing modules with different transactional semantics and reliability properties.

A preliminary account of the ideas underlying the transactor model was published as [9]; it contained no correctness proofs. While the work presented here shares some of the ideas of the earlier paper, almost all of the semantic components of τ-calculus have been updated and simplified.

3. TRANSACTOR MODEL

The goal of the transactor model is to enable developing reliable systems composed from potentially unreliable components, which may suffer both system failures and application-specific semantic inconsistencies. We show that given any two checkpointed global states k and k' of a distributed system such that k and k' are related by an execution trace containing inconsistent states resulting from node failures, application-level failures, and lost messages, there exists an equivalent execution trace containing only message losses. Hence programmers using our model need only reason about the possibility of lost messages, not about the other forms of failure.

Transactors extend the actor model [1] by explicitly modeling node failures, network failures, persistent storage, and state immutability. A transactor encapsulates state and communicates with other transactors via asynchronous message passing. In response to a message, a transactor may create new transactors, send messages to other transactors, or modify its internal state. In addition to these inherited actor operations, a transactor may stabilize, checkpoint, or rollback.

A transactor's stabilization is a commitment not to modify its internal state—i.e., to become immutable—until a subsequent checkpoint is performed or until another peer actor causes it to rollback due to semantic inconsistencies. Stabilization can be thought of as the first phase of a two-phase commitment protocol.

A checkpoint serves two purposes: first, it is a commitment to make the current transactor state persistent, i.e. able to survive local temporary node failures; and second and most important, it is a consistency guarantee, i.e. there are no pending dependencies on the volatile state of peer transactors. *Dependence information* is carried along with messages, so that only globally consistent states can be checkpointed. Checkpoint can be thought of as the second phase of a two-phase commitment protocol.

A rollback operation brings a transactor back to its previously checkpointed state, if any, or makes it disappear otherwise. Node failures have a similar effect.

4. THE TAU CALCULUS

The τ-calculus is based on an extended, untyped, call-by-value lambda calculus; its terms are depicted in Fig. 3. The basic lambda calculus constructs are standard and we will not comment on them further. The extensions can be divided into two categories, those terms ($\mathcal{E}_A$) that encode the traditional actor [1] semantics with explicit state management, and additional constructs to support distributed state maintenance. In this section, we will give a brief, intuitive tour of τ-calculus constructs, and defer a more detailed discussion of its semantics to Section 6.

4.1 Traditional Actor Constructs

The *transactor creation* construct $\mathbf{trans}\ e_1\ \mathbf{init}\ e_2\ \mathbf{snart}$ creates a new transactor with *behavior* e_1, and initial state e_2. The behavior must evaluate to an abstraction term; intuitively, this term evaluates each incoming message to the created transactor.

$\mathcal{A}$	$=$	$\{\mathsf{true}, \mathsf{false}, \mathsf{nil}, \ldots\}$	*Atoms*
$\mathcal{N}$	$=$	$\{0, 1, 2, \ldots\}$	*Natural numbers*
$\mathcal{T}$	$=$	$\{t_1, t_2, t_3, \ldots\}$	*Transactor names*
$\mathcal{X}$	$=$	$\{x_1, x_2, x_3, \ldots\}$	*Variable names*
$\mathcal{F}$	$=$	$\{=, +, \ldots\}$	*Primitive operators*
$\mathcal{V}$	$::=$	*Values*	
		$\mathcal{A} \mid \mathcal{N} \mid \mathcal{T} \mid \mathcal{X}$	
	$\mid$	$\lambda\mathcal{X}.\,\mathcal{E}$	*Lambda abstraction*
	$\mid$	$\langle\mathcal{V}, \mathcal{V}\rangle$	*Pair constructor*
$\mathcal{E}_P$	$::=$	*Pure expressions*	
		$\mathcal{V}$	
	$\mid$	$(\mathcal{E}\ \mathcal{E})$	*Lambda application*
	$\mid$	$\mathbf{fst}(\mathcal{E})$	*First element of pair*
	$\mid$	$\mathbf{snd}(\mathcal{E})$	*Second element of pair*
	$\mid$	$\mathbf{if}\ \mathcal{E}\ \mathbf{then}\ \mathcal{E}\ \mathbf{else}\ \mathcal{E}\ \mathbf{fi}$	*Conditional*
	$\mid$	$\mathbf{letrec}\ \mathcal{X} = \mathcal{E}\ \mathbf{in}\ \mathcal{E}\ \mathbf{ni}$	*Recursive definition*
	$\mid$	$\mathcal{F}(\mathcal{E}, \ldots, \mathcal{E})$	*Primitive operator*
$\mathcal{E}_A$	$::=$	*Traditional actor constructs*	
		$\mathcal{E}_P$	
	$\mid$	$\mathbf{trans}\ \mathcal{E}\ \mathbf{init}\ \mathcal{E}\ \mathbf{snart}$	*New transactor*
	$\mid$	$\mathbf{send}\ \mathcal{E}\ \mathbf{to}\ \mathcal{E}$	*Message send*
	$\mid$	$\mathbf{ready}$	*Ready to receive message*
	$\mid$	$\mathbf{self}$	*Reference to own name*
	$\mid$	$\mathbf{setstate}(\mathcal{E})$	*Set transactor state*
	$\mid$	$\mathbf{getstate}$	*Retrieve transactor state*
$\mathcal{E}$	$::=$	*Transactor expressions*	
		$\mathcal{E}_A$	
	$\mid$	$\mathbf{checkpoint}$	*Make failure-resilient*
	$\mid$	$\mathbf{rollback}$	*Revert to prev. checkpt.*
	$\mid$	$\mathbf{stabilize}$	*Prevent state changes*
	$\mid$	$\mathbf{dependent}?$	*Test dependence*

Figure 3: Terms.

The expression returns a *transactor name*, a fresh value that can be subsequently used as the target of the *message send* construct, $\mathbf{send}\ v\ \mathbf{to}\ t$. This construct sends a message with *contents* v to the transactor named t. The $\mathbf{ready}$ construct indicates that a transactor is waiting to process the next incoming message. $\mathbf{self}$ yields the transactor's own name.

The $\mathbf{setstate}(v)$ construct imperatively updates a transactor's state to the value v. A message send can potentially introduce a causal dependency from the sender to the target transactor, if the target transactor modifies its state in response to the message. When a transactor has committed not to change its state, execution of $\mathbf{setstate}(v)$ has no effect; thus the expression $\mathbf{setstate}(v)$ returns a boolean value indicating whether or not the state update actually took place. $\mathbf{getstate}$ retrieves the value of the state.

4.2 State Maintenance Constructs

The $\mathbf{stabilize}$ construct causes the current transactor to ignore subsequent $\mathbf{setstate}(v)$ or $\mathbf{rollback}$ expressions and become *stable*; this fact is communicated by the underlying operational semantics to other transactors with which the current transactor corresponds, and is in effect a "promise" to the transactor's peers that the transactor will not attempt to change its own state. Note that even after entering a *stable* state via the $\mathbf{stabilize}$ construct, a transactor can still process messages, it simply cannot change its own state.

The $\mathbf{checkpoint}$ construct creates a *checkpoint*, which is (effectively) a copy of the transactor's current state which can be recovered in the event of certain failures. A $\mathbf{checkpoint}$ can only be made if the current transactor is not *dependent* on the volatile state of one or more other transactors. That is, a state potentially unrecoverable in the presence of node failures. The $\mathbf{dependent}?$ construct tests whether this is the case. The $\mathbf{rollback}$ construct causes a transactor to revert to its previous checkpoint, if one exists, and causes the transactor to disappear otherwise. As with the $\mathbf{setstate}(v)$ construct, the **rollback** construct has no effect when the transactor is stable.

4.3 Defined Forms

Fig. 4 depicts a number of defined forms that provide convenient syntactic sugar for writing τ-calculus programs. Most of these constructs are self explanatory, a few deserve further explanation:

The **msgcase** construct yields a lambda abstraction whose body processes incoming messages. Messages are assumed to take the form of a vector of parameters, the first of which is an atom that constitutes a *message name*. The **msgcase** body tests the value of the incoming message and processes the other message arguments appropriately; messages that are not understood are ignored.

The **declstate** construct declares names for a transactor's state, which is presumed to consist of a vector of elements. This construct does not "expand" into a core τ-calculus expression, instead, it simply defines a static name scope for subsequent references of the form $!u$ and $u := e$, which expand into appropriate operations on the transactor's state vector.

```
[vec1]  $\langle e_1, \ldots, e_n \rangle$          $\triangleq$  $\langle e_1, \langle \ldots, \langle e_n, nil \rangle \rangle \ldots \rangle, \quad n > 0$
[vec2]  $\langle\rangle$                           $\triangleq$  $\mathsf{nil}$
[vec3]  $\vec{x}$                                  $\triangleq$  $\langle x_1, \ldots, x_n \rangle$   for some $n \geq 0$
[seq]   $e_1; e_2$                                 $\triangleq$  $((\lambda x.\ e_2)\ e_1), \quad x \notin fv(e_2)$
[if1]   $\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{fi}$   $\triangleq$  $\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ \mathsf{nil}\ \mathbf{fi}$
[let1]  $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{ni}$  $\triangleq$  $((\lambda x.\ e_2)\ e_1)$
[let2]  $\mathbf{let}\ \langle x_1, \ldots, x_n \rangle = e_1\ \mathbf{in}$
          $e_2$
        $\mathbf{ni}$
          $\triangleq$  $\mathbf{let}\ x_1 = \mathbf{fst}(e_1)\ \mathbf{in}$
                  ...
                    $\mathbf{let}\ x_n = \mathbf{fst}(\mathbf{snd}(\ldots \mathbf{snd}(e_1) \ldots))\ \mathbf{in}$
                      $e_2$
                    $\mathbf{ni}$
                  ...
                $\mathbf{ni}$
[vabs]  $\lambda \vec{x}.\ e$
          $\triangleq$  $\lambda x'.\ \mathbf{let}\ \vec{x} = x'\ \mathbf{in}\ e\ \mathbf{ni}, \quad x' \notin fv(e)$
[msg1]  $\mathsf{msg}\ \vec{x}$                    $\triangleq$  $\langle \mathsf{msg}, \vec{x} \rangle$
[msg2]  $\mathbf{msgcase}$
          $\mathsf{msg}_1\ \vec{x}_1 \ \Rightarrow\ e_1$
        | ...
        | $\mathsf{msg}_n\ \vec{x}_n \ \Rightarrow\ e_n$
        $\mathbf{esac}$
          $\triangleq$  $\lambda \langle m, z' \rangle.$ (
                  $\mathbf{if}\ m = \mathsf{msg}_1\ \mathbf{then}$
                    $\mathbf{let}\ \vec{x}_1 = z'\ \mathbf{in}\ e_1\ \mathbf{ni}$
                  $\mathbf{else}$
                    ...
                    $\mathbf{if}\ m = \mathsf{msg}_n\ \mathbf{then}$
                      $\mathbf{let}\ \vec{x}_n = z'\ \mathbf{in}\ e_n\ \mathbf{ni}$
                    $\mathbf{else}$
                      $\mathbf{ready}$
                    $\mathbf{fi}$
                    ...
                  $\mathbf{fi}$;
                  $\mathbf{ready}$ )
[sta1]  $\mathbf{declstate}\ \langle u_1, \ldots, u_n \rangle\ \mathbf{in}\ e\ \mathbf{etats}$
          $\triangleq$  declaration of names for $n$ elements of state
[sta2]  $!u_i$
          $\triangleq$    $\mathbf{let}\ \langle x_1, \ldots, x_i, \vec{z} \rangle = \mathbf{getstate}\ \mathbf{in}\ x_i\ \mathbf{ni}$
                  $u_i$ is the $i$th name declared in the closest
                  statically-enclosing $\mathbf{declstate}$ scope, of length $n$, $n \geq i > 0$
[sta3]  $u_i := e$
          $\triangleq$    $\mathbf{setstate}(\langle !u_1, \ldots, !u_{i-1}, e, !u_{i+1}, \ldots, !u_n \rangle)$
                  where $u_i$ is the $i$th name declared in the closest
                  statically-enclosing $\mathbf{declstate}$ scope, of length $n$, $n \geq i > 0$
```

Figure 4: Defined forms.

5. TRANSACTOR EXAMPLES

In this section, we illustrate a few representative transactor programs.

5.1 Reference Cells

We begin with a simple reference cell and two *reliable* versions thereof providing progressively more refined notions of consistent state under different failure and interaction assumptions.

The *cell* program, shown in Figure 5, is a volatile reference cell that never gets checkpointed: it cannot tolerate process failures and therefore, any other programs which depend on that cell's value will not be able to reach consistent states or checkpoint.

The $pcell_1$ program is a first attempt to provide a cell whose invariant is to always checkpoint its current value to be able to recover from process failures. Upon creation it must receive an initialize message, that creates an initial checkpoint. Notice that it first needs to become stable to succeed checkpointing. Also notice that the creator of that cell needs to be stable as well for that checkpoint to succeed. On reception of a set message, the cell will modify its value, and checkpoint again. This checkpoint assumes that the transactor sending the set message is stable, and therefore, does not create spurious dependencies on the cell upon state assignment. On reception of a get message, the cell needs to stabilize first, to ensure that no new dependencies are incurred in the cell's customer. And finally, to preserve the invariant of being just checkpointed (and therefore, volatile) on message reception, it does a final **checkpoint**.

The $pcell_2$ program builds on the previous example, but also considers the possibility that the clients setting the value of the cell may do it from a volatile (i.e., unstable) state. In this case, the cell's set message handler checks for any outstanding dependencies after updating its state, and if the transactor is dependent on other transactors, it rolls back to its previously (known to be) consistent state. $pcell_2$ is strictly more reliable than $pcell_1$ in the sense that it considers interaction with potentially volatile clients.

5.2 Electronic Money Transfer

The traditional electronic money transfer example depicted in Fig. 6 is implemented with transactors using a a protocol similar to classical two-phase commit protocols. *teller* represents an ATM machine or a similar coordinator for a transfer between two *bankaccount*s. All of the transactors are assumed to be persistent initially, and we assume that the *teller* has exclusive access to both accounts. Isolation and locking, if needed to ensure exclusive access, can be managed by appropriate auxiliary transactors.

The basic protocol used in the example is quite simple. The *teller* sends appropriate account adjustment requests to each account. Each account separately determines whether it is able to fulfill the request. If so, it stabilizes and sends done (with a result message) to the *teller*. If not, it also sends done (with an error message) to the teller, then rolls back. When the teller has received two done messages, it stabilizes, then requests that each account send a ping message both to its peer account and to the teller. Note that at this point in the protocol, the *teller* has no idea whether the update has been successful or not (assuming that it is not interpreting the messages returned by done). However, if either of the transactors has rolled back in the meantime due to insufficient funds or spontaneous failure, the ping messages will incorporate inconsistent dependence information, thus effectively resulting in rollback when received. In the absence of failure, each transactor will eventually receive sufficient ping messages for the **checkpoint** operation to

```
let cell = trans
  declstate ⟨contents⟩ in
    msgcase
      set⟨val⟩ ⇒
        contents := val
    | get⟨customer⟩ ⇒
        send data⟨!contents⟩
          to customer
    esac
  etats
init
  ⟨0⟩
snart
```

```
let pcell_1 = trans
  declstate ⟨contents⟩ in
    msgcase
      initialize⟨⟩ ⇒
        stabilize;
        checkpoint
    | set⟨val⟩ ⇒
        contents := val;
        stabilize;
        checkpoint
    | get⟨customer⟩ ⇒
        stabilize;
        send data⟨!contents⟩
          to customer;
        checkpoint
    esac
  etats
init
  ⟨0⟩
snart
```

```
let pcell_2 = trans
  declstate ⟨contents⟩ in
    msgcase
      initialize⟨⟩ ⇒
        stabilize;
        checkpoint
    | set⟨val⟩ ⇒
        contents := val;
        if dependent? then
          rollback
        else
          stabilize;
          checkpoint
        fi
    | get⟨customer⟩ ⇒
        stabilize;
        send data⟨!contents⟩
          to customer;
        checkpoint
    esac
  etats
init
  ⟨0⟩
snart
```

Figure 5: A progressively more refined reference cell. The leftmost example is an unreliable reference cell. The middle one is a persistent reference cell which assumes stable clients. The rightmost cell represents a persistent reliable reference cell.

succeed; until that point, the **checkpoint** is a no-op.

The protocol in Fig. 6 ensures that the transfer will always complete in a consistent state, either with both accounts updated appropriately, or both left unchanged. The protocol does not deal directly with certain combinations of message losses; however, it could easily be augmented by adding a *timer* transactor that periodically re-sends ping requests if the participants have not checkpointed.

Note that we could easily interpose a *currencyconverter* transactor between participants *which does not need to know that the parties involved are part of a transaction*—the model enables to compose services with full transaction semantics with services that do not have any transactional behavior in a seamless and correct manner.

5.3 Web Application Server

The *appserver* example in Fig. 7 models a web application server. The application server provides a newsession operation that creates a new session transactor for a given customer. These session transactors will be completely independent from each other, i.e., no dependencies will be introduced by the application server. *db* is a reference cell representing a database resource, which can be volatile or persistent in nature–see reference cell examples above. While the database is shared across multiple web application customers, the customer sessions are independent of each other.

The application server does not depend on the customer transactors it interacts with, since its state is read-only –similarly, to a currency conversion service in the bank account example. If the application server is properly initialized, it will also not induce any dependencies on the created sessions or the interacting customers.

6. OPERATIONAL SEMANTICS

In this section, we provide an operational semantics for the τ-calculus. We first need to establish some notational conventions.

6.1 Notational Preliminaries

Most of the notation we use in the sequel is standard or self-explanatory. Here, we cover a few concepts that are not standard.

```
let appserver = trans
  declstate ⟨db⟩ in
    msgcase
      initialize⟨⟩ ⇒
        stabilize
    | newsession⟨cust⟩ ⇒
        send
          trans
            declstate ⟨customer, db⟩ in
              msgcase
                setdata⟨val⟩ ⇒
                  send set⟨val⟩ to !db
              | getdata⟨⟩ ⇒
                  send get⟨!customer⟩ to !db
              esac
            etats
          init
            ⟨cust, !db⟩
          snart
        to cust
    esac
  etats
init
  ⟨db⟩
snart
```

Figure 7: Web application server illustrating use of a stable "daemon" to create instances of per-session transactors. Since the server is stable, the session transactors can act autonomously with no dependence on the server.

Grammars as sets. We will often define sets using context-free grammars, and will use a non-terminal of the grammar to represent the set of all terms derivable from that non-terminal.

Lists. Given a set S, we will use $[S]$ to denote the set of *lists* defined over S, where $[]$ denotes the empty list, and $s :: l_s$ denotes a list cell. We will frequently use $[e_1; e_2; \ldots ; e_n]$ as a shorthand to denote $e_1 :: (e_2 :: (\ldots (e_n :: []) \ldots))$. $len(l)$ denotes the length of l, and $lastn(n, l)$ denotes the list consisting of the last n elements of l (for $0 \leq n \leq len(l)$).

```
let bankaccount = trans
  declstate ⟨bal⟩ in
    msgcase
      adj⟨delta, atm⟩ ⇒
        bal := !bal + delta;
        if !bal < 0 then
          send done⟨"Not enough funds!"⟩ to atm
          rollback
        else
          stabilize;
          send done⟨"Balance update successful"⟩ to atm
        fi
    | pingreq⟨requester⟩ ⇒
        send ping⟨⟩ to requester
    | ping⟨⟩ ⇒    // may cause rollback
        checkpoint
    esac
  etats
init
  ⟨0⟩
snart
```

```
let teller = trans
  declstate ⟨inacct, outacct, acks⟩ in
    msgcase
      transfer⟨delta⟩ ⇒
        send adj⟨delta, self⟩ to !inacct;
        send adj⟨−delta, self⟩ to !outacct
    | done⟨msg⟩ ⇒
        send println⟨msg⟩ to stdout;
        acks := !acks + 1;
        if !acks = 2 then
          stabilize;
          send pingreq⟨!inacct⟩ to !outacct;
          send pingreq⟨!outacct⟩ to !inacct;
          send pingreq⟨self⟩ to !outacct;
          send pingreq⟨self⟩ to !inacct
        fi
      | ping⟨⟩ ⇒    // may cause rollback
          checkpoint
      esac
    etats
  init
    ⟨savings, checking, 0⟩
  snart
```

Figure 6: Electronic money transfer example. Illustrates nontrivial use of stabilize for a protocol similar to two-phase commit. Note that ping messages are used to communicate status (stable or rolled-back) implicitly: checkpoints resulting from receipt of ping messages will succeed only if all peer transactors have stabilized; otherwise it will be a no-op (if pings have not yet been received from peers), or cause rollback (if peer is inconsistent).

Finite maps. Given sets S_1 and S_2, $S_1 \xrightarrow{f} S_2$ denotes the set of finite partial maps from S_1 to S_2, where $dom(m)$ and $ran(m)$ denote the domain and range of m, respectively. We will use $\emptyset$ to denote the empty map, $m(x)$ to denote the element to which m maps x, $m[x \mapsto e]$ to denote the map that is the same as m, except that x is mapped to e, and $m \setminus x$ to denote the map m' that is the same as m, except that $x \notin dom(m')$. We will use $[x \mapsto e]$ as a shorthand for $\emptyset[x \mapsto e]$. Let m be a map, and f be a function from $ran(m)$ to $ran(m)$. Then we will use $m[x \mapsto f]$ as a shorthand for the map $m[x \mapsto f(m(x))]$. $m'(x) = f(m(x))$. If we want to apply f to selected elements of a map, we will sometimes use "map comprehension" expressions such as $\{[x \mapsto f(e)] \mid x \in dom(m), e = m(x), p(x, e)\}$ to generate new maps from m in the obvious way.

Multisets. If $\mathcal{S}$ is a set, then $\{\{\mathcal{S}\}\}$ denotes the set of multisets, (i.e., bags) consisting of collections of elements of $\mathcal{S}$. We will use '$\uplus$' to denote multiset union. We will also sometimes use "multiset comprehension" expressions such as $\{\{f(x) \mid x \in \mathcal{M}, p(x)\}\}$ to generate new multisets from $\mathcal{M}$ in the obvious way (multiple instances of x generate the same number of instances of $f(x)$). We will use $s \setminus x$ to denote the multiset s' which is the same as s, except that one instance of x has been removed.

Pattern Matching. When writing rules comprising the operational semantics for transactors, we will use various *pattern matching* constructs, both to determine the applicability of a particular rule, and to match components of terms to variables. In addition to the usual convention of building patterns by applying term constructors to variables, we will also use the following additional pattern-related conventions: The underscore character '_' matches any term. The pattern $m[x \mapsto p]$ matches any map m' for which $x \in dom(m')$ and p matches $m'(x)$; the variable m is then bound to the map $m' \setminus x$. Finally, the pattern $s \uplus \{x\}$ matches any multiset s', in which case x is bound to an *arbitrary* element of s', and s is bound to the multiset $s' \setminus x$.

6.2 Reduction Contexts

Each transition rule of our operational semantics will refer to a particular *redex* term within the lambda term encoding a transactor's behavior. As is standard for lambda calculi, we will use the notion of *reduction contexts* of the form $\mathcal{R}(\Box)$ to distinguish the redex on which the transition rule will operate. Each reduction context is a special term with a single "hole" element $\Box$, defined such that a transactor behavior can be uniquely decomposed into exactly one redex and one reduction context. The redexes and reduction contexts are depicted in Fig. 8.

6.3 Transactor Configurations

Fig. 9 depicts a collection of semantic domains that the τ-calculus operational semantics will manipulate.

A *volatility value* $w \in \mathcal{W}$ encodes the fact that a transactor is *volatile* ($w = \mathbf{V}(n)$ for some $n \geq 0$) or *stable* ($w = \mathbf{S}(n), n \geq 0$). The value of n will be referred to as an *incarnation*.

A *history* $h \in \mathcal{H}$ encodes the checkpoint history of a transactor. A history $h = \langle w, l_h \rangle$ encodes the fact that the transactor which refers to h has volatility value w, and has checkpointed $len(l_h)$ times since its creation, where the values in the list l_h reflect the incarnation at which each checkpoint occurred.

The τ-calculus semantics defines four operations on histories:

1. When a transactor is created, its history is initialized to $\langle \mathbf{V}(0), [] \rangle$.
2. When a transactor with history $\langle \mathbf{V}(n), l_h \rangle$ rolls back, its incarnation is incremented by 1, i.e., its history becomes $\langle \mathbf{V}(n+1), l_h \rangle$.
3. If a transactor with history $\langle \mathbf{S}(n), l_h \rangle$ checkpoints, its history becomes $\langle \mathbf{V}(0), n :: l_h \rangle$.
4. If a transactor with history $\langle \mathbf{V}(n), l_h \rangle$ stabilizes, its history becomes $\langle \mathbf{S}(n), l_h \rangle$.

Dependence maps Δ are critical auxiliary structures that can informally be thought of as encoding the states of all transactors on

$$
\begin{array}{lll}
\mathcal{E}_P^{rdx} & ::= & \textit{Pure redexes} \\
 & & (\mathcal{V}\ \mathcal{V}) \\
 & | & \mathbf{fst}(\mathcal{V}) \\
 & | & \mathbf{snd}(\mathcal{V}) \\
 & | & \mathbf{if}\ \mathcal{V}\ \mathbf{then}\ \mathcal{E}\ \mathbf{else}\ \mathcal{E}\ \mathbf{fi} \\
 & | & \mathbf{letrec}\ \mathcal{X} = \mathcal{V}\ \mathbf{in}\ \mathcal{E}\ \mathbf{ni} \\
 & | & \mathcal{F}(\mathcal{V}, \ldots, \mathcal{V}) \\
 & & \\
\mathcal{E}^{rdx} & ::= & \textit{Redexes} \\
 & & \mathcal{E}_P^{rdx} \\
 & | & \mathbf{self} \\
 & | & \mathbf{dependent}? \\
 & | & \mathbf{setstate}(\mathcal{V}) \\
 & | & \mathbf{getstate} \\
 & | & \mathbf{checkpoint} \\
 & | & \mathbf{stabilize} \\
 & | & \mathbf{rollback} \\
 & | & \mathbf{ready} \\
 & | & \mathbf{trans}\ \mathcal{V}\ \mathbf{init}\ \mathcal{V}\ \mathbf{snart} \\
 & | & \mathbf{send}\ \mathcal{V}\ \mathbf{to}\ \mathcal{V} \\
 & & \\
\mathcal{R} & ::= & \textit{Reduction contexts} \\
 & & \Box \\
 & | & (\mathcal{R}\ \mathcal{E}) \\
 & | & (\mathcal{V}\ \mathcal{R}) \\
 & | & \langle \mathcal{R}, \mathcal{E} \rangle \\
 & | & \langle \mathcal{V}, \mathcal{R} \rangle \\
 & | & \mathbf{fst}(\mathcal{R}) \\
 & | & \mathbf{snd}(\mathcal{R}) \\
 & | & \mathbf{if}\ \mathcal{R}\ \mathbf{then}\ \mathcal{E}\ \mathbf{else}\ \mathcal{E}\ \mathbf{fi} \\
 & | & \mathbf{letrec}\ \mathcal{X} = \mathcal{R}\ \mathbf{in}\ \mathcal{E}\ \mathbf{ni} \\
 & | & \mathcal{F}(\mathcal{V}, \ldots, \mathcal{V}, \mathcal{R}, \mathcal{E}, \ldots, \mathcal{E}) \\
 & | & \mathbf{setstate}(\mathcal{R}) \\
 & | & \mathbf{trans}\ \mathcal{R}\ \mathbf{init}\ \mathcal{E}\ \mathbf{snart} \\
 & | & \mathbf{trans}\ \mathcal{V}\ \mathbf{init}\ \mathcal{R}\ \mathbf{snart} \\
 & | & \mathbf{send}\ \mathcal{R}\ \mathbf{to}\ \mathcal{E} \\
 & | & \mathbf{send}\ \mathcal{V}\ \mathbf{to}\ \mathcal{R}
\end{array}
$$

Figure 8: Redexes and reduction contexts.

which some value depends. More precisely, a dependence map maps each transactor name t on which it is defined to a history value associated with t. Dependence maps are associated with three distinct semantic components of a transactor: the transactors on which t is dependent for its *existence* (the *creation* dependence map), the transactors on which t's current state depends (the *state* dependence map), and the transactors on which the value of the current redex depends (the *behavioral* dependence map). By separating a transactor's dependences into three components, we can distinguish those dependences related to creation from those related to state (which have radically different semantic consequences), and avoid the creation of spurious dependences when, e.g., a transactor never reads its state.

A *transactor* $\tau \in \mathcal{S} = \langle b, s_{\surd}\ ;\ e, s\ ;\ \delta_s, \delta_c, \delta_b \rangle$ is a 7-tuple containing the following components: The *volatile state* component s contains τ's current state; we say that s is volatile since its value is lost in the event of τ's failure. By contrast, the *persistent state* component $s_{\surd}$ encodes the last value of s stored by a **checkpoint** operation; this state is resilient to failure and models stable storage. The *behavior* of τ, i.e., its fixed response to every incoming message, is represented by b; in order to be well-formed, b must be a lambda expression. The *evaluation state* component, e, is an expression (generally partially evaluated) representing the current state of the evaluation of a transactor's behavior. The *state dependence map* component, δ_s, is a dependence map that encodes the fact that the state of τ is dependent (transitively) on the states of all of the transactors in $dom(\delta_s)$, whose histories are encoded in the map. The *creation dependence map* component, δ_c, is similar to δ_s, except that it records information about the transitive dependence of τ on the *parent* transactor that initially created τ. Finally, the *behavioral dependence map* component, δ_b, represents the *behavioral dependences* of the transactor, i.e., the dependences of the current redex under evaluation.

$\mathcal{W}$	$::=$	$\mathbf{V}(\mathcal{N}) \mid \mathbf{S}(\mathcal{N})$	*Volatility value*
$\mathcal{H}$	$::=$	$\langle \mathcal{W}, [\mathcal{N}] \rangle$	*Transactor history*
Δ	$=$	$\mathcal{T} \xrightarrow{f} \mathcal{H}$	*Dependence map*
$\mathcal{S}$	$::=$	$\langle \mathcal{V}, \mathcal{V}\ ;\ \mathcal{E}, \mathcal{V}\ ;\ \Delta, \Delta, \Delta \rangle$	*Transactor*
$\mathcal{M}$	$::=$	$\mathcal{T} \Leftarrow \langle \mathcal{V}, \Delta \rangle$	*Message*
Θ	$=$	$\mathcal{T} \xrightarrow{f} \mathcal{S}$	*Name service*
$\mathcal{K}$	$::=$	$\{\{\mathcal{M}\}\} \mid \Theta$	*Transactor configuration*

Figure 9: Semantic domains.

Note that we use both commas and semicolons to separate components of a transactor. There is no semantic distinction between the two; this is a purely syntactic convention designed to separate transactor components into three (semicolon-separated) "logical clusters" for easier reading. These clusters represent, respectively, persistent (i.e., durable) components that survive failures (b and $s_{\surd}$), volatile components that generally do not survive failures (e and s), and dependence information (δ_s, δ_c, and δ_b).

A *message* $m \in \mathcal{M}$ contains a target transactor name encoding the message's destination, a value representing the message's *payload*, and a dependence map encoding the transitive closure of transactors on which the message's payload is dependent.

A transactor *configuration* $k \in \mathcal{K}$ is a pair consisting of a *network*, a multiset of messages, and a *nameserver* map from transactor names to transactors. The network serves to buffer messages sent among the transactors in the configuration. The multiset representation for the network encodes the fact that the order in which messages sent to the same transactor are received is unrelated to the order in which they were sent (*even from the same sender*).

6.4 History and Dependence Map Operations

In this section, we define a number of auxiliary operations on histories, dependence maps, and related structures that will be required by the operational semantics.

Basic history operations. We begin by defining some basic operations on histories. Let $h = \langle w, l_h \rangle$ be a history. Then h is *stable*, notated $\Diamond(h)$, if $w = \mathbf{S}(n)$ for some n; otherwise h is *volatile*. If l_h is nonempty, i.e., it has checkpointed, then h is *persistent*, notated $\surd(h)$; otherwise, h is *ephemeral*. The *empty history* $\langle \mathbf{V}(0), [] \rangle$ will be denoted by $\mathbf{H}_0$.

Relations on histories. Next, we define some relations on histories that will be used in the transition rules in the operational semantics for the τ-calculus. '$\looparrowright$', '$\rightarrow_{\Diamond}$', and '$\rightarrow_{\surd}$' are the least relations satisfying the following conditions:

$$
\begin{array}{rcll}
\langle \mathbf{V}(n), l_h \rangle & \looparrowright & \langle \mathbf{V}(n+1), l_h \rangle & \text{("rolls back to")} \\
\langle \mathbf{S}(n), l_h \rangle & \looparrowright & \langle \mathbf{V}(n+1), l_h \rangle & \text{("rolls back to")} \\
\langle \mathbf{V}(n), l_h \rangle\rangle & \rightarrow_{\Diamond} & \langle \mathbf{S}(n), l_h \rangle & \text{("stabilizes to")} \\
\langle \mathbf{S}(n), l_h \rangle & \rightarrow_{\surd} & \langle \mathbf{V}(0), n :: l_h \rangle & \text{("checkpoints to")}
\end{array}
$$

'$\looparrowright$', '$\rightarrow_{\Diamond}$', and '$\rightarrow_{\surd}$' represent all of the valid "single-step" transitions that a history associated with a single transactor can make: '$\looparrowright$' encodes the fact that a transactor has rolled back. A volatile transactor can roll itself back (the first case for '$\looparrowright$') or be rolled back "spontaneously" due to node failure or inconsistent state; a stable transactor (the second case) only rolls back if its state is found to be inconsistent. The '$\rightarrow_{\Diamond}$' transition encodes the fact that a transactor has stabilized, and '$\rightarrow_{\surd}$' encodes the fact that a transactor has checkpointed. Since these relations are functions, we will sometimes speak of "applying" them to a history to yield a new history.

We can now define the following composite relation:

$$\rightsquigarrow \quad \triangleq \quad (\looparrowright \cup \rightarrow_{\Diamond} \cup \rightarrow_{\surd}) \qquad \text{("is succeeded by")}$$

Intuitively, $h_1 \rightsquigarrow_* h_2$ if h_1 and h_2 are valid histories for the same transactor (say, τ), and h_2 occurs after h_1 in some execution trace for τ. '$\rightsquigarrow_*$' defines a partial order on histories. We will say that histories h_1 and h_2 are *comparable* if either $h_1 \rightsquigarrow_* h_2$ or $h_2 \rightsquigarrow_* h_1$.

Finally, we have the following relation:

$$\rtimes \quad \triangleq \quad \looparrowright \cdot \rightsquigarrow_* \qquad \text{("is superseded by")}$$

Intuitively, $h_1 \rtimes h_2$ if h_2 is a history of a transactor that rolled back from the state represented by history h_1, then proceeded to do zero or more additional operations. Thus the state represented by h_2 supersedes the obsolete state represented by h_1. We will say that two histories are *consistent* if neither supersedes the other.

Given consistent histories h_1 and h_2 we define the *sharpening* operation, notated $h_1 \sharp h_2$, as follows:

$$h_1 \sharp h_2 \quad = \quad \begin{cases} h' & \text{if there exists } h' \text{ such that} \\ & \quad h_1 \rightarrow_{\Diamond} h' \rightsquigarrow_* h_2 \\ h_1 & \text{otherwise} \end{cases}$$

Intuitively, if h_1 is not stable, and h_2 is reachable (via '$\rightsquigarrow_*$') from h_1 via an intermediate history h' which is the stable form of h_1, then the sharpening operation yields h', otherwise it is a no-op. The sharpening operation is used to "update" dependence information about peer transactors that have stabilized since their last communication.

Operations on dependence maps. Let δ_1 and δ_2 be dependence maps. Then δ_1 is *invalidated by* δ_2, notated $\delta_1 \rtimes \delta_2$ if and only if there exists t such that $t \in dom(\delta_1) \cap dom(\delta_2)$ and $\delta_1(t) \rtimes \delta_2(t)$.

Let δ_1 and δ_2 be dependence maps. Then their *union*, denoted $\delta_1 \oplus \delta_2$, is defined as follows:

$$(\delta_1 \oplus \delta_2)(t) = \begin{cases} \max_{\rightsquigarrow_*}(\delta_1(t), \delta_2(t)) & \\ & \text{when } t \in dom(\delta_1) \cap dom(\delta_2) \text{ and} \\ & \quad \delta_1(t) \text{ and } \delta_2(t) \text{ are comparable} \\ \delta_1(t) & \text{when } t \in dom(\delta_1),\ t \notin dom(\delta_2) \\ \delta_2(t) & \text{when } t \notin dom(\delta_1),\ t \in dom(\delta_2) \\ \text{undef.} & \text{otherwise} \end{cases}$$

We extend the sharpening operation on histories to consistent dependence maps δ_1 and δ_2 as follows:

$$(\delta_1 \sharp \delta_2)(t) \quad = \quad \begin{cases} \delta_1(t) \sharp \delta_2(t) & \text{when} \\ & \quad t \in dom(\delta_1) \cap dom(\delta_2) \\ \delta_1(t) & \text{otherwise} \end{cases}$$

Let δ be a dependence map. Then δ is *independent*, notated $\Diamond(\delta)$ if for all $t \in dom(\delta)$, $\Diamond(\delta(t))$; otherwise, δ is *dependent*.

Characterizing transactors. Let

$$\tau = \langle b, s_{\surd} \,;\, e, s \,;\, \delta_s[t \mapsto h], \delta_c, \delta_b \rangle$$

be a transactor bound to name t in some transactor configuration Then we will say that τ is *stable* if $\Diamond(h)$ and *volatile* otherwise. Transactor τ is *independent* if $\Diamond(\delta_s \oplus \delta_c \setminus t)$ (i.e., τ depends on no unstable transactors other than itself) and *dependent* otherwise. Transactor τ is *ready* if $e = \mathbf{ready}$, and *busy* otherwise; it is *persistent* if $\surd(h)$ and *ephemeral* otherwise; it is *initial* if $s = s_{\surd}$ and non-initial otherwise. If τ is persistent, independent, ready, and initial, we will say that it is *resilient*.

When not otherwise qualified, we will refer to the volatile state of τ, i.e., s, as simply the *state of* τ. If τ is both stable and independent, we will say that it is a *daemon*. Daemons can be used to model humans or other "external agents" in a system that send and receive messages, are resilient to (system!) failure, but do not "participate" in global state.

Operations on configurations. Let $k = \mu \mid \theta$ be a configuration. Then we will use $net(k)$ to denote the network μ, and $ns(k)$ to denote the nameserver θ. The *domain* of k, denoted by $dom(k)$ is the set of all transactors in k's name service map, i.e., $dom(ns(k))$. Given a configuration k and a transactor name $t \in dom(k)$, we will use $k(t)$ as a shorthand for the transactor $(ns(k))(t)$. We will say that a configuration k is *ready* or *resilient* iff for all $t \in dom(k)$, $k(t)$ is ready or resilient, respectively.

6.5 Transactor Configuration Transition Rules

We will divide the transition rules of the τ-calculus into two principal classes: those representing *normal* transitions, where the only form of failure allowed is message loss, and *node failure* transitions representing either spontaneous node failures or rules designed to manage inconsistencies resulting from such failures.

The set of normal transitions will be represented by the composite transition relation '$\underset{\smile}{\longrightarrow}$', which is the relational union of the primitive transition rules in Fig. 11 and 12. The transition rules in Figs. 11 encode the "classical" semantics of the Actor model [1]. The transition rules in Fig. 12 augment the classical semantics with additional operations for managing consistency (e.g., creating checkpoints).

The set of node failure transitions will be represented by the composite transition relation '$\underset{\frown}{\longrightarrow}$', which is the relational union of the primitive transition rules in all of Figs. 13 and 14. The transition rules in Figure 13 model "spontaneous" node failures; i.e., failures beyond the control of the transactors themselves. The transition rules in Figure 14 define the semantics of "program-induced" failures via the **rollback** operation, and other operations to handle inconsistencies resulting from failures.

We will use '$\xrightarrow[\tau]{}$' to denote an arbitrary τ-calculus transition, i.e., $\xrightarrow[\tau]{} = \underset{\smile}{\longrightarrow} \cup \underset{\frown}{\longrightarrow}$

In the following sections, we will consider each collection of rules in turn. While the number of transition rules may appear somewhat daunting initially, we believe that each of them encodes a "semantically orthogonal" component of τ-calculus semantics in a reasonably natural way.

Pure Reduction Rules. Fig. 10 depicts a set of standard *pure* reduction rules for lambda terms encoding transactor behaviors. These rules are "imported" into the classical actor calculus whose transition rules are depicted in Fig. 11.

Transition Rules for Basic Actor Semantics. Fig. 11 depicts the collection of transition rules that encode the semantics of the Actor model [1]. The semantics is loosely modeled after the semantics of Agha et al.[3], but with a significantly different treatment of state. In the rules of Fig. 11 as well as other rules in the sequel, the relation $\xrightarrow[l]{t}$ is a *single-step* transition relation on transactor configurations. Single-step transitions will be annotated with both the name of the applicable rule and a distinguished transactor name t to which the relation will be said to *apply*. Given a transactor configuration k that maps transactor name t to transactor τ,

$$\begin{array}{lrcl}
[\mathrm{pur1}] & ((\lambda x.\ e)\ v) & \longrightarrow_\lambda & e[v/x] \\
[\mathrm{pur2}] & \mathbf{fst}(\langle v_1, \_\rangle) & \longrightarrow_\lambda & v_1 \\
[\mathrm{pur3}] & \mathbf{snd}(\langle \_, v_2\rangle) & \longrightarrow_\lambda & v_2 \\
[\mathrm{pur4}] & \mathbf{if}\ \mathsf{true}\ \mathbf{then}\ e_1\ \mathbf{else}\ \_\ \mathbf{fi} & \longrightarrow_\lambda & e_1 \\
[\mathrm{pur5}] & \mathbf{if}\ \mathsf{false}\ \mathbf{then}\ \_\ \mathbf{else}\ e_2\ \mathbf{fi} & \longrightarrow_\lambda & e_2 \\
[\mathrm{pur6}] & \mathbf{letrec}\ x = v\ \mathbf{in}\ e\ \mathbf{ni} & \longrightarrow_\lambda & \\
& \multicolumn{3}{r}{e[(v[(\mathbf{letrec}\ x = v\ \mathbf{in}\ e\ \mathbf{ni})/x])/x]} \\
[\mathrm{pur7}] & f(v_1, \ldots, v_n) & \longrightarrow_\lambda & v \\
& \multicolumn{3}{r}{(f \in \mathcal{F}, v = [\![f]\!](v_1, \ldots, v_n))}
\end{array}$$

Figure 10: Pure reduction rules.

it will be convenient to refer to τ by its name, t. We now consider each rule in turn.

$[\mathrm{pure}]$ This rule applies one of the pure reduction rules depicted in Fig. 10 to the behavior of a transactor.

$[\mathrm{new}]$ This rule creates a new transactor t' with behavior b' and initial state s'. The persistent state is initially nil since t' has not yet checkpointed. The state dependence map for t' is initialized to refer to itself: a transactor is always dependent on itself (while this information may appear to be redundant, it avoids technical problems when a transactor sends messages to itself, which among other things is a convenient way to encode "continuations" to be performed following checkpoints). The creation dependence map for t' is the dependence map union of the creation dependences and behavioral dependences for the creating transactor and a mapping for the creating transactor (t) itself. This map encodes those transactors on whose states t''s creation is transitively dependent. Note that the behavioral dependence map for t is updated to encode a dependence on the newly-created transactor. This "contravariant" dependence is critical for ensuring that the persistent state of a transactor cannot refer to an ephemeral (i.e., noncheckpointed) transactor.

$[\mathrm{send}]$ This rule encodes the act of sending message m with payload v_m to transactor t_2. The message is "tagged" with the creation and behavioral dependences of the sender (transactor t_1), as well as a dependence on t_1 itself. Thus m carries information about the transactors on which it is *transitively* dependent. Note that it is not necessary to incorporate the state dependences of the sender; those are included in the behavioral dependence map if the state is ever read.

$[\mathrm{rcv1}]$ This rule encodes message receipt. Note that messages are selected from the network component of a configuration nondeterministically. Thus while our model assumes guaranteed message delivery, it does not guarantee order of delivery. The preconditions of the rule ensure that no received message either invalidates the state or creation of the receiver t, nor is invalidated by t. The preconditions always hold in the absence of failures; rules addressing the failure of these preconditions are addressed below. As a result of message receipt, the behavioral dependences of t are updated to contain the dependences of the received message, and t's behavior (a lambda expression) is applied to the message. Finally, t's current state and creation dependence maps are updated by the sharpening operation ('$\sharp$') to reflect new information about those dependences contained in the arriving message. In particular, we need to determine if any previously volatile transactors on which t is dependent have now become stable.

$[\mathrm{get}]$, $[\mathrm{set1}]$ These rules model retrieving and setting state, which we model as a single (possibly composite) cell. Note that in the case of rule $[\mathrm{get}]$, the updated behavioral dependence map δ'_b encodes a dependence on the state, symmetrically, rule $[\mathrm{set1}]$ adds information in the behavioral dependence map to update the state dependence map, δ'_s. Among other things, this semantics ensures that if a transactor t does not update its state in the course of processing a message from another transactor t' on which t was not previously dependent, t will not become dependent on t'.

$[\mathrm{self}]$ This rule encodes retrieval of the transactor's own name.

Core Transactor Transition Rules.

The rules depicted in Fig. 12 augment the basic actor transitions of Fig. 11 with additional rules for managing distributed state, as follows:

$[\mathrm{set2}]$ This rule causes the expression $\mathbf{setstate}(v)$ to be ignored when target transactor t is stable; this encodes a "promise" to peer transactors that t will not voluntarily update its state or roll back (however, it may nonetheless be rolled back due to inconsistencies).

$[\mathrm{sta1}]$, $[\mathrm{sta2}]$ These rules encode the stabilization operation. Stabilization inhibits further state updates and $\mathbf{rollback}$ operations (via rules $[\mathrm{set2}]$ and $[\mathrm{rol1}]$), renders the transactor resilient to spontaneous failure (due to the absence of rules for such failures in Fig. 13), and is a prerequisite to checkpointing (rule $[\mathrm{chk1}]$). Rule $[\mathrm{sta1}]$ applies if the transactor is currently volatile; it simply updates the transactor's history to reflect the fact that it is stable. Rule $[\mathrm{sta2}]$ encodes the fact that stabilization is a no-op if the transactor is already stable.

$[\mathrm{chk1}]$, $[\mathrm{chk2}]$ These rules encode the $\mathbf{checkpoint}$ operation. The preconditions of rules $[\mathrm{chk1}]$ and $[\mathrm{chk2}]$ determine whether t has received messages from all of the transactors on which it is dependent indicating that those transactors have stabilized or checkpointed the relevant dependent states. If the checkpoint operation succeeds (rule $[\mathrm{chk1}]$), the volatile state of t is stored in t's persistent state, t's history is updated to reflect the checkpoint, and the state dependences of t are reset, and the creation and behavioral dependence maps are reset to $\emptyset$. Thus in addition to storing volatile state persistently, the dependence map resetting performed by the checkpoint operation has the effect of bounding the amount of dependence information that must be tracked across checkpoints. The resetting of the creation dependence map to $\emptyset$ implies that this map is only non-empty for ephemeral transactors. If the preconditions for checkpointing do not hold, rule $[\mathrm{chk2}]$ causes it to behave like $\mathbf{ready}$.

$[\mathrm{rol1}]$ Rule $[\mathrm{rol1}]$ encodes the fact that programmatic rollback is disallowed when t is stable; in this case, rollback behaves as if it were $\mathbf{ready}$.

$[\mathrm{dep1}]$, $[\mathrm{dep2}]$ These rules determine whether t is dependent on any non-stable transactors other than itself.

$[\mathrm{lose}]$ Finally, this rule models the fact that under "normal circumstances" messages may be lost after being sent. We assume that such losses are relatively rare; however it may initially seem odd to make message loss an element of normal transactor behavior at all. In part, this is a consequence of our global consistency semantics, which trades the possibility of global inconsistency for the possibility of message loss, hence transforming the programmer's burden from reasoning about *global* failures (about which they can have no knowledge in general), to reasoning about a *local* failure in the form of lost messages. However, as a practical matter, even programs running in systems with guarantees about message delivery must *effectively* reason about the possibility of message loss, since they typically must incorporate time-outs to deal with protracted message latencies (which then become indistinguishable from losses).

Failure Transitions.

The rules depicted in Fig. 13 model *spontaneous* node failure caused by faults. In realistic systems, these rules will be applied far less frequently than the non-failure rules.

$[\mathrm{fl1}]$ This rule models the transient node failure of a persistent, volatile transactor. In such cases, the state of the transactor reverts

$[\mathrm{pure}]$ *Evaluate pure redex.*

$$\frac{e \longrightarrow_{\lambda} e' \quad e \in \mathcal{E}_P^{rdx}}{\mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\,e\,], s\ ;\ \delta_s, \delta_c, \delta_b\rangle] \xrightarrow[\mathrm{[pure]}]{t} \mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\,e'\,], s\ ;\ \delta_s, \delta_c, \delta_b\rangle]}$$

$[\mathrm{new}]$ *Create new transactor.*

$$\begin{array}{ll} \mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\,\mathbf{trans}\ b'\ \mathbf{init}\ s'\ \mathbf{snart}\,], s\ ;\ \delta_s[t \mapsto h], \delta_c, \delta_b\rangle] & t' \notin dom(\theta) \cup \{t\} \\ \xrightarrow[\mathrm{[new]}]{t} \quad \mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\,t'\,], s\ ;\ \delta_s[t \mapsto h], \delta_c, \delta_b \oplus \delta'\rangle][t' \mapsto \langle b', \mathsf{nil}\ ;\ \mathbf{ready}, s'\ ;\ \delta', \delta_c \oplus \delta_b \oplus [t \mapsto h], \emptyset\rangle] & \delta' = [t' \mapsto \mathbf{H}_0] \end{array}$$

$[\mathrm{send}]$ *Send message, piggybacking dependence information.*

$$\mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\,\mathbf{send}\ v_m\ \mathbf{to}\ t'\,], s\ ;\ \delta_s[t \mapsto h], \delta_c, \delta_b\rangle] \xrightarrow[\mathrm{[send]}]{t} (\mu \uplus \{t' \Leftarrow \langle v_m, \delta_c \oplus \delta_b \oplus [t \mapsto h]\rangle\}) \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\,\mathsf{nil}\,], s\ ;\ \delta_s[t \mapsto h], \delta_c, \delta_b\rangle]$$

$[\mathrm{rcv1}]$ *Message dependences not invalidated by transactor; transactor dependences not invalidated by message: process message normally.*

$$\frac{\neg(\delta_{sc} \rtimes \delta_m) \qquad \neg(\delta_m \rtimes \delta_{sc})}{(\mu \uplus \{t \Leftarrow \langle v_m, \delta_m\rangle\}) \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\,\mathbf{ready}\,], s\ ;\ \delta_s, \delta_c, \_\rangle] \xrightarrow[\mathrm{[rcv1]}]{t} \mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ (b\ v_m), s\ ;\ \delta_s \sharp\, \delta_m, \delta_c \sharp\, \delta_m, \delta_m\rangle]} \qquad \delta_{sc} = \delta_s \oplus \delta_c$$

$[\mathrm{get}]$ *Retrieve state.*

$$\mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\,\mathbf{getstate}\,], s\ ;\ \delta_s, \delta_c, \delta_b\rangle] \xrightarrow[\mathrm{[get]}]{t} \mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\,s\,], s\ ;\ \delta_s, \delta_c, \delta_b \oplus \delta_s\rangle]$$

$[\mathrm{set1}]$ *Transactor is volatile: setting state succeeds.*

$$\frac{\neg\Diamond(h)}{\mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\,\mathbf{setstate}(s)\,], \_\ ;\ \delta_s[t \mapsto h], \delta_c, \delta_b\rangle] \xrightarrow[\mathrm{[set1]}]{t} \mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\,\mathsf{true}\,], s\ ;\ \delta_s[t \mapsto h] \oplus \delta_b, \delta_c, \delta_b\rangle]}$$

$[\mathrm{self}]$ *Yields reference to own name.*

$$\mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\,\mathbf{self}\,], s\ ;\ \delta_s, \delta_c, \delta_b\rangle] \xrightarrow[\mathrm{[self]}]{t} \mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\,t\,], s\ ;\ \delta_s, \delta_c, \delta_b\rangle]$$

Figure 11: Transition rules encoding basic actor semantics.

to the stored persistent state, and the state dependence information is reinitialized. This rule assumes that a persistent transactor is capable of checkpointing intermediate states to stable storage, then restoring such checkpoints after a failure (e.g., following a reboot or software recovery).

[fl2] This rule models the permanent node failure of an ephemeral transactor: it is annihilated. This rule models systems that cannot checkpoint intermediate states to stable storage; these systems are assumed to fail by stopping permanently.

Note that if a transactor is stable, no failure rule applies. This means in practice that the "program counter" for intermediate evaluation states of a stable transactor's behavior must be logged to persistent storage. While this may seem like a rather onerous requirement, we expect that the number of intermediate states in computations performed by a stable transactor will be minimal. Also, many optimizations are possible to minimize the overhead of this requirement in practice, e.g., deferring all "side effects" (message sends or transactor creations) to cause them to be executed during a (local) ACID transaction of short duration.

Transactor Rules for Managing Inconsistency. The final collection of rules, depicted in Fig. 14, encode programmatic rollback and manage the inconsistencies that result from explicit rollback or inconsistencies due to incoming messages. The inconsistency management rules are as follows:

[rol2], [rol3] These rules (along with [rol1]) above encode the **rollback** operation. Rule [rol1] encodes the fact that programmatic rollback is disallowed when t is stable; in this case, rollback behaves as if it were **ready**. Rule [rol2] encodes the fact that if an ephemeral (non-checkpointed) transactor rolls back, it disappears, i.e., is *annihilated* (among other things, this behavior allows certain transactors to "dispose of themselves" when their work is done). Otherwise, rule [rol3] encodes the fact that rollback resets the (volatile) state to the last stored persistent state; in addition, the state, creation, and behavioral dependences are reinitialized.

[rcv2] This rule applies when the dependences associated with an incoming message are *invalidated* by the state or creation dependences associated with t. This occurs if the message depends on an earlier incarnation of some dependent transactor than t does. In this case, the message is ignored to ensure global consistency.

[rcv3] This rule applies when the dependences associated with an incoming message m *supersede* the state dependences (but not the creation dependences) associated with t, and t is persistent. In such cases, t is effectively rolled back to ensure global consistency, and the result is the same as in rule [rol3].

[rcv4] This rule applies when an ephemeral transactor's state or creation dependences are invalidated by an incoming message. In this case, t cannot roll back since there is no checkpoint to roll back to; instead, it is annihilated to ensure global consistency.

7. FORMAL PROPERTIES

In this section, we define what it means for a system such as the τ-calculus to be well-behaved. In particular, we prove certain soundness and liveness properties appropriate for the τ-calculus. For soundness, we show that a trace (i.e., a transition sequence) containing node failures and inconsistencies is equivalent to a normal trace, i.e., one containing no node failures, but possibly message losses. We also show that checkpointing is *possible*, assuming certain reasonable preconditions. First, we need some preliminary definitions.

7.1 Preliminary Definitions

Relations. We will define a number of relations to facilitate reasoning about successive states of transactor configurations, individual transactors, or components thereof. To define these relations, we will use the following relational operators: Given binary relations R, R_1 and R_2, $R*$ denotes the reflexive, transitive closure

[set2] *Transactor is stable: attempt to set state fails.*

$$\frac{\Diamond(h)}{\mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathcal{R}[\,\mathbf{setstate}(\_)\,], s \,;\, \delta_s[t \mapsto h], \delta_c, \delta_b\rangle] \xrightarrow[\text{[set2]}]{t} \mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathcal{R}[\,\mathsf{false}\,], s \,;\, \delta_s[t \mapsto h], \delta_c, \delta_b\rangle]}$$

[sta1] *Transactor is volatile: stabilization causes it to become stable.*

$$\frac{h \to_{\Diamond} h'}{\mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathcal{R}[\,\mathbf{stabilize}\,], s \,;\, \delta_s[t \mapsto h], \delta_c, \delta_b\rangle] \xrightarrow[\text{[sta1]}]{t} \mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathcal{R}[\,\mathsf{nil}\,], s \,;\, \delta_s[t \mapsto h'], \delta_c, \delta_b\rangle]}$$

[sta2] *Transactor currently stable:* **stabilize** *is a no-op.*

$$\frac{\Diamond(h)}{\mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathcal{R}[\,\mathbf{stabilize}\,], s \,;\, \delta_s[t \mapsto h], \delta_c, \delta_b\rangle] \xrightarrow[\text{[sta2]}]{t} \mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathcal{R}[\,\mathsf{nil}\,], s \,;\, \delta_s[t \mapsto h], \delta_c, \delta_b\rangle]}$$

[chk1] *Transactor is stable and independent: checkpoint succeeds.*

$$\frac{\Diamond(\delta_s[t \mapsto h] \oplus \delta_c) \qquad h \to_{\surd} h'}{\mu \mid \theta[t \mapsto \langle b, \_ \,;\, \mathcal{R}[\,\mathbf{checkpoint}\,], s \,;\, \delta_s[t \mapsto h], \delta_c, \_\rangle] \xrightarrow[\text{[chk1]}]{t} \mu \mid \theta[t \mapsto \langle b, s \,;\, \mathbf{ready}, s \,;\, [t \mapsto h'], \emptyset, \emptyset\rangle]}$$

[chk2] *Transactor is dependent or volatile:* **checkpoint** *simply behaves like* **ready**.

$$\frac{\neg\Diamond(\delta_s \oplus \delta_c)}{\mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathcal{R}[\,\mathbf{checkpoint}\,], s \,;\, \delta_s, \delta_c, \_\rangle] \xrightarrow[\text{[chk2]}]{t} \mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathbf{ready}, s \,;\, \delta_s, \delta_c, \emptyset\rangle]}$$

[rol1] *Transactor is stable:* **rollback** *simply behaves like* **ready**.

$$\frac{\Diamond(h)}{\mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathcal{R}[\,\mathbf{rollback}\,], s \,;\, \delta_s[t \mapsto h], \delta_c, \_\rangle] \xrightarrow[\text{[rol1]}]{t} \mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathbf{ready}, s \,;\, \delta_s[t \mapsto h], \delta_c, \emptyset\rangle]}$$

[dep1] *Transactor is independent: yields false.*

$$\frac{\Diamond((\delta_s \oplus \delta_c) \setminus t)}{\mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathcal{R}[\,\mathbf{dependent?}\,], s \,;\, \delta_s, \delta_c, \delta_b\rangle] \xrightarrow[\text{[dep1]}]{t} \mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathcal{R}[\,\mathsf{false}\,], s \,;\, \delta_s, \delta_c, \delta_b\rangle]}$$

[dep2] *Transactor is dependent: yields true.*

$$\frac{\neg\Diamond((\delta_s \oplus \delta_c) \setminus t)}{\mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathcal{R}[\,\mathbf{dependent?}\,], s \,;\, \delta_s, \delta_c, \delta_b\rangle] \xrightarrow[\text{[dep2]}]{t} \mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathcal{R}[\,\mathsf{true}\,], s \,;\, \delta_s, \delta_c, \delta_b\rangle]}$$

[lose] *Message loss.*

$$(\mu \uplus \{m\}) \mid \theta \xrightarrow[\text{[lose]}]{m} \mu \mid \theta$$

Figure 12: Transition rules encoding basic transactor semantics.

[fl1] *Spontaneous failure of volatile, persistent transactor causes rollback.*

$$\frac{\surd(h) \qquad \neg\Diamond(h) \qquad h \looparrowright h'}{\mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \_, \_ \,;\, \delta_s[t \mapsto h], \delta_c, \delta_b\rangle] \xrightarrow[\text{[fl1]}]{t} \mu \mid \theta[t \mapsto \langle b, s_{\surd} \,;\, \mathbf{ready}, s_{\surd} \,;\, [t \mapsto h'], \emptyset, \emptyset\rangle]}$$

[fl2] *Spontaneous failure of volatile, ephemeral transactor causes it to be annihilated.*

$$\frac{\neg\surd(h) \qquad \neg\Diamond(h)}{\mu \mid \theta[t \mapsto \langle \_, \mathsf{nil} \,;\, \_, \_ \,;\, \delta_s[t \mapsto h], \_, \_\rangle] \xrightarrow[\text{[fl2]}]{t} \mu \mid \theta}$$

Figure 13: Transition rules modeling spontaneous failures.

of R; R^{-1} denotes the relational inverse of R; and $R_1 \cup R_2$ and $R_1 \cdot R_2$ denote respectively the relational union and relational composition of R_1 and R_2. If R is a binary relation on elements of a set $\mathcal{S}$, then we will say that $s \in \mathcal{S}$ is an R *normal form* if there exists no $s' \in \mathcal{S}$ such that $s \; R \; s'$. If $s \; R{*} \; s'$ and s' is an R normal form, then we will say that s' is an R normal form *of* s (or *the* R normal form if s' is unique).

Traces. If $S = \{R_1, R_2, \ldots, R_m\}$ is a set of binary relations and $R = R_1 \cup R_2 \cup \ldots \cup R_m$, we will refer to R' as a *composite relation* based on the *basis set* S of *primitive relations*. In general, primitive relations will represent "single step" transition relations for an operational semantics. If S is a basis set of primitive relations such that for all $R_1, R_2 \in S$, $R_1 \cap R_2 = \emptyset$, we will say that S is an *orthogonal* basis set. Let $S = \{R_1, R_2, \ldots, R_n\}$ be a set of primitive relations, and R' be the composite relation based on S. Then we will refer to a (possibly empty) sequence of primitive relations from the set S as an R'*-trace*. Given an initial value x_0 and an R'-trace $\rho = R_{i_1} \; R_{i_2} \; \ldots \; R_{i_m}$ over an orthogonal basis set, there exists a unique sequence $x_0 \; x_1 \; \ldots \; x_m$ such that

$$x_0 \; (R_{i_1} \cdot R_{i_2} \cdots R_{i_m}) \; x_m$$

In this case, we will use the trace ρ to refer *either* to the sequence of relations $R_{i_1} R_{i_2} \ldots R_{i_m}$ *or* the sequence of values $x_0 \; x_1 \; \ldots \; x_{m-1}$, and will also feel free to treat ρ as the set of values $\{x_0, x_1, \ldots, x_{m-1}\}$ when convenient. Note that we adopt the convention that the value sequence represented includes the initial element of the transition sequence, but not the final element. We

[rol2] *Transactor is volatile and ephemeral: rollback causes transactor to be annihilated.*

$$\frac{\neg\Diamond(h) \qquad \neg\surd(h)}{\mu \mid \theta[t \mapsto \langle \_, \mathsf{nil}\ ;\ \mathcal{R}[\ \mathbf{rollback}\ ], \_\ ;\ \delta_s[t \mapsto h], \_, \_\rangle] \quad \xrightarrow[\text{[rol2]}]{t} \quad \mu \mid \theta}$$

[rol3] *Transactor is volatile and persistent: rollback reverts state to contents of persistent state saved by last checkpoint.*

$$\frac{\neg\Diamond(h) \qquad \surd(h) \qquad h \looparrowright h'}{\mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\ \mathbf{rollback}\ ], \_\ ;\ \delta_s[t \mapsto h], \_, \_\rangle] \quad \xrightarrow[\text{[rol3]}]{t} \quad \mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathbf{ready}, s_{\surd}\ ;\ [t \mapsto h'], \emptyset, \emptyset\rangle]}$$

[rcv2] *Message dependences invalidated by those of transactor but not vice-versa: discard message.*

$$\frac{\delta_m \rtimes \delta_{sc} \qquad \neg(\delta_{sc} \rtimes \delta_m)}{(\mu \uplus \{t \Leftarrow \langle \_, \delta_m\rangle\}) \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\ \mathbf{ready}\ ], s\ ;\ \delta_s, \delta_c, \_\rangle] \quad \xrightarrow[\text{[rcv2]}]{t} \quad \mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathbf{ready}, s\ ;\ \delta_s, \delta_c, \emptyset\rangle]} \qquad \delta_{sc} = \delta_s \oplus \delta_c$$

[rcv3] *State dependences (but not creation dependences) invalidated by message and transactor is persistent: transactor rolls back.*

$$\frac{\delta_s[t \mapsto h] \rtimes \delta_m \qquad \neg(\delta_c \rtimes \delta_m) \qquad \surd(h) \qquad h \looparrowright h'}{\mu \uplus \{t \Leftarrow \langle v_m, \delta_m\rangle\} \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathcal{R}[\ \mathbf{ready}\ ], \_\ ;\ \delta_s[t \mapsto h], \delta_c, \_\rangle] \quad \xrightarrow[\text{[rcv3]}]{t} \quad \mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathbf{ready}, s_{\surd}\ ;\ [t \mapsto h'], \emptyset, \emptyset\rangle]}$$

[rcv4] *State or creation dependences invalidated by message and transactor is ephemeral: transactor is annihilated.*

$$\frac{\delta_{sc} \rtimes \delta_m \qquad \neg\surd(h)}{(\mu \uplus \{t \Leftarrow \langle \_, \delta_m\rangle\}) \mid \theta[t \mapsto \langle \_, \mathsf{nil}\ ;\ \mathcal{R}[\ \mathbf{ready}\ ], \_\ ;\ \delta_s[t \mapsto h], \delta_c, \_\rangle] \quad \xrightarrow[\text{[rcv4]}]{t} \quad \mu \mid \theta} \qquad \delta_{sc} = \delta_s[t \mapsto h] \oplus \delta_c$$

Figure 14: Transition rules for programmatic rollback and consistency management.

will frequently use the notation $x_0 \xrightarrow{\rho}_* x_m$ when ρ is an S-trace, and $\longrightarrow$ is the composite relation based on S. We will use ϵ to denote an empty trace, and $len(\rho)$ to denote the length of a trace ρ.

Configuration well-formedness. In this section, we define what it means for a transactor configuration to be "sensible" with respect to its history annotations. Let $\tau = \langle b, s_{\surd}; e, s; \delta_s, \delta_c, \delta_b\rangle$ be a transactor, and let t' be an arbitrary transactor name. Then the set of *histories of* t' *associated with* τ is denoted by $histories(t', \tau)$, defined by

$$histories(t', \tau) \triangleq \\ \{h' \mid \delta_s(t') = h' \ \text{ or } \ \delta_c(t') = h' \ \text{ or } \ \delta_b(t') = h'\}$$

Note that this set is *not* necessarily a singleton; e.g., τ's creation can be dependent on one checkpointed version of t', and its current state on a different version.

Let μ be a network, and t be an arbitrary transactor name. Then the set of *histories of* t *associated with* μ is denoted by $histories(t, \mu)$, defined by

$$histories(t, \mu) \triangleq \{h \mid (\_ \Leftarrow \langle \_, \delta_m[t \mapsto h]\rangle) \in \mu\}$$

Let k be a well-formed transactor configuration, and t be a transactor such that $t \in dom(k)$. Then the *principal history* of t in k is denoted by $history(t, k)$, and is defined by

$$history(t, k) \triangleq \\ \delta_s(t) \quad \text{such that} \quad k(t) = (\langle b, s_{\surd}\ ;\ e, s\ ;\ \delta_s, \delta_c, \delta_b\rangle)$$

Let k be a configuration. Then the set of *t-dependent node histories in k* is denoted by $depHists(t, k)$, and is defined by

$$depHists(t, k) = \bigcup_{t' \in (dom(k) \setminus t)} histories(t', k(t))$$

Thus $depHists(t, k)$ yields the set of all histories of t present in nodes of k with the exception of its principal history. Given configuration k, a transactor t in $dom(k)$ is *garbage* if $t \notin depHists(t', k)$ for any other transactor t'.

We will say that a configuration k is *well-formed* iff the following conditions hold:

1. For all $t \in dom(k)$ such that $k(t) = (\langle b, s_{\surd}\ ;\ e, s\ ;\ \delta_s, \delta_c, \delta_b\rangle)$, $t \in dom(\delta_s)$, and if $\surd(history(t, k))$, then $\delta_c = \emptyset$.
2. For all $h \in depHists(t, k)$, $h \rightsquigarrow_* history(t, k)$

In other words, for a configuration to be well-formed, every transactor t must have its own history in its state dependence map and its creation dependence map must be empty if t has checkpointed. In addition, a transactor's principal history must be the "most recent" of all the histories of t associated with other transactors in k.

LEMMA 1 (WELL-FORMEDNESS PRESERVATION). *Let k be a well-formed configuration, and let k' be a configuration such that $k \xrightarrow[\tau]{\rho}_* k'$. Then k' is also well-formed.*

PROOF. Straightforward induction on $len(\rho)$. □

Configuration consistency. In this section, we define notions of *consistency* for transactor configurations. Inconsistent configurations will correspond to transactors whose states are inconsistent due to node failures. Let $\tau = \langle b, s_{\surd}\ ;\ e, s\ ;\ \delta_s, \delta_c, \delta_b\rangle$ be a transactor. Then the *composite dependence map* for τ, notated $maps(\tau)$ is defined by $maps(\tau) \triangleq \delta_s \oplus \delta_c \oplus \delta_b$. Let k be a configuration, and $t \in dom(k)$ be a transactor name. Then the *composite dependence map* for t in k, notated $maps(t, k)$, is defined by $maps(t, k) = maps(k(t))$.

Given a configuration k, we will say that a transactor $t \in dom(k)$ is *consistent* (with respect to k) if there exists no t' in $dom(k)$ such that $maps(t, k)(t') \rtimes history(t', k)$. In other words, k is dependent on no other transactor t' for which the state of t' is currently inconsistent with t. Similarly, a message $(t \Leftarrow \langle \_, \delta_m\rangle) \in ns(k)$ is *consistent* (with respect to k) if there exists no t' in $dom(k)$ such that $\delta_m(t') \rtimes history(t', k)$.

We will say that nameserver θ is *consistent* if for all $t \in dom(\theta)$, t is consistent. A well-formed network μ is *consistent* if for all $m \in \mu$, m is consistent. A configuration k is *network consistent* if $net(k)$ is consistent with respect to k and *node consistent* if $ns(k)$ is consistent with respect to k. Finally, a configuration k is *consistent* if it is both network consistent and node consistent.

Configuration equivalence modulo history. In this section, we define a simple notion of transactor equivalence that is oblivious to certain inconsequential differences in dependence information. Given two histories h and $\hat{h}$, such that $\hat{h}$ is a predecessor history to h, the *reversion* operation $revert_{\hat{h}}(h)$ defines a new history h' that is "the same" as h, except that the operations represented by $\hat{h}$ do not occur:

$$revert_{\hat{h}}(h) \quad = \quad \begin{cases} h' & \text{if there exists } \hat{h}_0 \text{ such that} \\ & \quad \hat{h}_0 \looparrowright \hat{h} \overset{\rho}{\rightsquigarrow}_* h \text{ and} \\ & \quad \hat{h}_0' \overset{\rho}{\rightsquigarrow}_* h' \\ h & \text{otherwise} \end{cases}$$

The definition above will be critical to defining a node-failure free trace from a corresponding trace with node failures: if $\hat{h}$ represents a set of failing operations in a transactor, we will "extract" those operations from a trace and update other histories h using $revert_{\hat{h}}(h)$.

Let t be a transactor name, h be a history, and k be a transactor configuration. Then $revert_h(t, k)$ is defined as follows:

$$\begin{array}{l} revert_h(t, \mu \mid \theta) \triangleq \mu' \mid \theta' \\ \quad \text{where} \\ \quad \mu' \quad = \quad \{\!\{\ (t' \Leftarrow \langle v_m, \delta_m[t \mapsto revert_h]\rangle) \\ \qquad\qquad\qquad \mid (t' \Leftarrow \langle v_m, \delta_m\rangle) \in \mu\ \}\!\} \\ \quad \text{and} \\ \quad \theta' \quad = \quad \{[\ t' \mapsto \langle \quad b, s_{\surd}\ ;\ e, s\ ; \\ \qquad\qquad\qquad\qquad \delta_s[t \mapsto revert_h], \\ \qquad\qquad\qquad\qquad \delta_c[t \mapsto revert_h], \\ \qquad\qquad\qquad\qquad \delta_b[t \mapsto revert_h] \quad \rangle\ ] \\ \qquad\qquad\quad \mid t' \in dom(\theta) \quad \text{and} \\ \qquad\qquad\qquad (\langle b, s_{\surd}\ ;\ e, s\ ;\ \delta_s, \delta_c, \delta_b\rangle) = \theta(t') \\ \qquad\qquad \} \end{array}$$

(Recall that $\delta_s[t \mapsto revert_h]$ is shorthand for the map $\delta_s[t \mapsto revert_h(\delta_s(t))]$; similarly for the other maps).

If ρ is a trace, we will use $revert_h(t, \rho)$ to denote the trace ρ' resulting from replacing every configuration $k \in \rho$ by $revert_h(t, k)$.

Let t be a transactor, k be t-consistent configuration, and $\hat{h} = history(t, k)$. Then $k \approx_{t,\hat{h}} k'$ if $k' = revert_{\hat{h}}(t, k)$. The relation '$\approx$', read "equivalence modulo history" is then defined as the least equivalence relation satisfying

$$k \approx_{t,h} k' \text{ for some } t, h \quad \Longrightarrow \quad k \approx k'$$

The relation '$\approx$' is a very weak form of configuration equivalence akin to α-equivalence in the lambda calculus or structural congruences in process calculi. The idea is that two configurations that are identical up to certain inconsequential differences in dependence information behave identically. This fact is embodied in the following lemma:

LEMMA 2 (BEHAVIOR OF $\approx$-EQUIVALENT CONFIGURATIONS). *Let k_1 and k_2 be configurations such that $k_1 \approx k_2$, and ρ be a trace such that $k_1 \xrightarrow[\tau]{\rho}_* k_1'$. Then there exists k_2' such that $k_2 \xrightarrow[\tau]{\rho}_* k_2'$ and $k_2 \approx k_2'$.*

PROOF. Straightforward induction on $len(\rho)$ and the definition of '$\approx$'. □

Cycle Properties. Let ρ be a $\xrightarrow[\tau]{}_*$ trace. Then a nonempty trace ρ is a *t-sequence* if all primitive transitions in ρ have the form $\xrightarrow[l]{t}$, i.e., all transitions are applicable to a transactor named t. A *cycle-terminating transition* is any primitive transition rule in $\xrightarrow[\tau]{}_*$ that either takes the form

$$\begin{array}{ll} \mu \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ e, s\ ;\ \delta_s, \delta_c, \delta_b\rangle] & \xrightarrow[l]{t} \\ \mu' \mid \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathbf{ready}, s\ ;\ \delta_s, \delta_c, \delta_b\rangle] & \end{array}$$

or

$$\mu \mid \theta[t \mapsto \tau] \quad \xrightarrow[l]{t} \quad \mu' \mid \theta$$

In other words, a cycle-terminating transition either causes a transactor's evaluation state to become **ready**, or results in the annihilation of some transactor. A t-trace c is a *t-cycle* if $c = c'\ r$ where c' is a $\xrightarrow[\tau]{}$ trace, and r is a cycle-terminating transition.

LEMMA 3 (CYCLE PROPERTIES).

1. *Every t-cycle contains exactly one transition from the set*

$$\{\xrightarrow[\text{[rcv1]}]{}, \xrightarrow[\text{[rcv2]}]{}, \xrightarrow[\text{[rcv3]}]{}, \xrightarrow[\text{[rcv4]}]{}, \xrightarrow[\text{[fl1]}]{}\}$$

   which must be the initial transition of the cycle.

2. *If a t-cycle contains a primitive transition in $\xrightarrow[\frown]{}$, then it can contain only one such transition, which must be the final transition in the cycle. In this case, we will refer to the cycle as a* failure *cycle.*

3. *If the last transition in a t-cycle is not an element of $\xrightarrow[\frown]{}$, then it must either be the* [pure] *transition or the* [chk1] *transition.*

4. *Let $c_i^{t_i}$ be a t-cycle. Then either*

   (a) For all $k_i \in c_i^{t_i}$, k_i is consistent, or

   (b) For all $k_i \in c_i^{t_i}$, k_i is inconsistent.

   Given this fact, we will refer to a t-cycle as either a consistent *cycle, or an* inconsistent *cycle.*

PROOF. Properties 1-3 follow trivially from the definitions of the primitive transitions. Property 4 follows from the previous properties and a simple induction on the length of a cycle. □

LEMMA 4 (CYCLE DECOMPOSITION). *Let k_1 and k_2 be well-formed and ready configurations such that $k_1 \xrightarrow[\tau]{\rho}_* k_2$. Then there exists a trace ρ' of the form*

$$\hat{\rho} = \lambda_0\ c_1^{t_1}\ \lambda_1\ \ldots\ c_n^{t_n}\ \lambda_n$$

where for all $1 \leq i \leq n$, $c_i^{t_i}$ is a t_i-cycle, and for all $0 \leq j \leq n$, λ_j is a (possibly empty) message loss trace *of the form $\xrightarrow[\text{[lose]}]{}_*$, such that $k_1 \xrightarrow[\tau]{\rho'}_* k_2$. We will refer to the trace $\hat{\rho}$ as a* cycle decomposition *of ρ.*

PROOF. By induction on $len(\rho)$. Define a total ordering on all transactor names present in ρ. Permute pairs of primitive non-loss transitions in ρ not consistent with the total ordering, and permute loss/non-loss pairs. The resulting trace has the desired form. □

7.2 Simulation Without Node Failures

Given the preceding definitions, we are now in a position to prove that arbitrary τ-calculus traces can be simulated by traces containing only the node failure free subset of the τ-calculus. We first require the following key lemma:

LEMMA 5 (SIMULATION). *Let k_1^α, k_2^α, k_1^β, and k_2^β be well-formed configurations, α and β be traces such that*

$$k_1^\alpha \xrightarrow[\tau]{\alpha}_* k_2^\alpha \quad \textit{and} \quad k_1^\beta \xrightarrow[\smile]{\beta}_* k_2^\beta$$

$T_{\looparrowright}$ and $T_{!}$ be sets of transactor names, and $M_{!}$ be a network (i.e., a multiset of messages). Assume k_1^α, k_2^α, k_1^β, k_2^β, α, β, $T_{\looparrowright}$, $T_{!}$, and $M_{!}$ all satisfy the following conditions:

1. *k_1^α, k_1^β, and k_2^β are resilient and network consistent.*
2. *For all $k \in \beta$, k is node consistent.*
3. *$T_{\looparrowright} \subseteq dom(k_1^\beta)$, and for all $t \in T_{\looparrowright}$, $history(t, k_2^\alpha) \looparrowright history(t, k_1^\beta)$ and $k_1^\beta(t)$ is initial.*
4. *For all $t \in dom(k_1^\beta) \setminus T_{\looparrowright}$, $k_2^\alpha(t) = k_1^\beta(t)$.*
5. *$T_{¡} \cup dom(k_1^\beta) = dom(k_2^\alpha)$, and for all $t \in T_{¡}$ such that $h = history(t, k_2^\alpha)$, $\neg\Diamond(h)$ and $\neg\surd(h)$.*
6. *$M_{¡} \uplus net(k_1^\beta) = net(k_2^\alpha)$, and for all $m \in M_{¡}$, m is inconsistent with respect to k_2^α.*

Then there exists configuration $k_2^{\beta'}$ and trace α' such that $k_2^{\beta'} \approx k_2^\beta$ and

$$k_1^\alpha \xrightarrow[\tau]{\gamma}{}_* \, k_2^{\beta'}$$

PROOF. Let

$$\hat{\alpha} = \lambda_0 \; c_1^{t_1} \; \lambda_1 \; \ldots \; c_n^{t_n}; \lambda_n$$

be a cycle decomposition of α. The the proof proceeds by induction on $len(\hat{\alpha})$, where at each inductive step, we either *remove* a cycle from the tail of α and re-establish the premises of the lemma (typically by updating the sets $T_{\looparrowright}$, $T_{¡}$, or $M_{¡}$ appropriately and sometimes by adding a $[\text{lose}]$ transition), or we show that we can *extend* the trace β by prepending the cycle to the trace β and re-establishing the premises of the lemma. The sets $T_{\looparrowright}$, $T_{¡}$, and $M_{¡}$ are used to allow certain differences to exist between configurations k_2^α and k_1^β at intermediate stages of the proof. These differences are "discharged" by the base case, at which point k_2^α and k_1^β become identical.

Base Case. If $\hat{\alpha} = \epsilon$, then since k_1^α is resilient, it is also node consistent. Since it is also network consistent by assumption, the other premises of the theorem can be satisfied if and only if $T_{\looparrowright} = T_{¡} = M_{¡} = \emptyset$ and $k_1^\alpha = k_2^\alpha = k_1^\beta$. In that case, we let $\gamma = \beta$ and the theorem follows immediately.

Inductive Case. Otherwise, we have

$$k_1^\alpha \xrightarrow[\tau]{\alpha'}{}_* \, k_2^{\alpha'} \xrightarrow[\tau]{c_n^{t_n} \lambda_n}{}_* \, k_2^\alpha$$

for some trace α' and configuration $k_2^{\alpha'}$. The proof has four main cases:

1. $c_n^{t_n}$ is a failure cycle.
2. $t_n \in T_{\looparrowright}$ and case 1 does not apply.
3. $c_n^{t_n}$ is an inconsistent cycle, and cases 1 and 2 do not apply.
4. None of cases 1–3 apply.

We will prove the result for an examplary subcase of Case 1 in full detail. The remaining cases use identical formal machinery; for those, we will provide more informal arguments, appealing to the same concepts covered in detail for Case 1.

Case 1. By Clause 2 of Lemma 3, the final transition in $c_n^{t_n}$ must be one of the rules in Figs. 13 and 14 and the initial transition must use the rule $[\text{rcv1}]$. We prove the result for rule [fl1], which is typical. The other rules use analogous reasoning.

Subcase [fl1]. By Clause 1 of Lemma 3, either [fl1] is the only transition in the cycle, or the first transition is an instance of the rule $[\text{rcv1}]$. We consider the latter case here; the former is a simpler subcase.

We proceed by removing cycle $c_n^{t_n}$; i.e., we show that there exist sets $T'_{\looparrowright}$, $T'_{¡}$, and $M'_{¡}$, configuration $k_1^{\beta'}$ and trace λ'_n such that

$$k_1^\alpha \xrightarrow[\tau]{\alpha'}{}_* \, k_2^{\alpha'} \quad \text{and} \quad k_1^{\beta'} \xrightarrow[\smile]{\lambda'_n \lambda_n}{}_* \, k_1^\beta \xrightarrow[\smile]{\beta'}{}_* \, k_2^{\beta'}$$

and such that $k_2^{\alpha'}$, $k_1^{\beta'}$, $k_2^{\beta'}$, and sets $T'_{\looparrowright}$, $T'_{¡}$, and $M'_{¡}$ satisfy the preconditions of the proof:

If any message m' is sent in cycle $c_n^{t_n}$, then due to the application of rule [fl1], it is inconsistent with respect to k_2^α. However, by the preconditions for the lemma, no inconsistent messages exist in configuration k_1^β, thus it must be the case that $m' \in M_{¡}$. To restore preconditions of the lemma after removing $c_n^{t_n}$, we set $M'_{¡} = M_{¡} \setminus m'$.

Let m be the message received during this cycle. If m were inconsistent with respect to configuration $k_2^{\alpha'}$, then a rule other than [fl1] would apply, hence m must be consistent. We therefore set $\lambda'_n = \xrightarrow[\text{[lose]}]{m}$ to restore precondition 6 of the lemma.

If any transactor t' is created in $c_n^{t_n}$, due to the application of rule [fl1], it is inconsistent with respect to k_2^α. Since by the preconditions of the lemma, no inconsistent transactors exist in configuration k_1^β, it must be the case that $t' \in T_{¡}$. To restore the precondition of the lemma after removing $c_n^{t_n}$, we therefore set $T'_{¡} = T_{¡} \setminus t'$.

If $t_n \in T_{\looparrowright}$ and $k_2^{\alpha'}(t_n)$ is initial, then by the definition of $T_{\looparrowright}$, $k_1^{\beta'}(t_n)$ is also initial, thus they have the same state and are identical modulo history information. We address this discrepancy as follows: By the definition of $T_{\looparrowright}$ and the fact that rule [fl1] rolls back t_n's history, we know that

$$history(t_n, k_2^{\alpha'}) \looparrowright history(t_n, k_2^\alpha) \looparrowright history(t_n, k_1^\beta)$$

Let $\hat{h} = history(t_n, k_2^\alpha)$ and $\beta' = revert_{\hat{h}}(t_n, \beta)$, in which case we have $k_2^{\beta'} = revert_{\hat{h}}(t_n, k_2^\beta)$, and hence $k_2^{\beta'} \approx k_2^\beta$. Also, we see that t_n still satisfies the condition for inclusion in $T_{\looparrowright}$, thus we set $T'_{\looparrowright} = T_{\looparrowright}$.

If $t_n \notin T_{\looparrowright}$, we set $T'_{\looparrowright} = T_{\looparrowright} \cup \{t_n\}$ to re-establish the premises of the lemma.

No transactors other than t_n are affected by the removal of $c_n^{t_n}$. Summarizing the construction above, we have

$$k_1^\alpha \xrightarrow[\tau]{\alpha'}{}_* \, k_2^{\alpha'} \quad \text{and} \quad k_1^{\beta'} \xrightarrow[\smile]{\lambda'_n \lambda_n \beta'}{}_* \, k_2^{\beta'}$$

where $T'_{\looparrowright}$, $T'_{¡}$, $M'_{¡}$, $k_2^{\alpha'}$, and $k_1^{\beta'}$ all satisfy the preconditions of the lemma. Since α' has length less than α, the result follows by induction.

Subcase [fl2]. This is very similar to the subcase for [fl1], except that t_n is annihilated, rather than being (effectively) rolled back. Therefore, rather than updating the set $T_{\looparrowright}$ to reflect the rolled-back history of t_n, we set $T'_{¡} = T_{¡} \cup \{t_n\}$ to reflect its annihilation. The rest of the construction is identical to that of the previous case.

Other Subcases. The remaining of the subcases of Case 1 use reasoning similar to the cases above, and exactly the same formal machinery.

Inductive Case 2. Since $t_n \in T_{\looparrowright}$, the history $history(t_n, k_1^\beta)$ invalidates $history(t_n, k_2^\alpha)$. Hence any message sent or transactor created in cycle $c_n^{t_n}$ will be inconsistent with configuration k_2^{β}. We restore the preconditions for the lemma by removing the cycle and define the sets $T'_{\looparrowright}$, $T'_!$, and $M'_!$ and trace β' based on updates of $T_{\looparrowright}$, $T_!$, $M_!$, and β analogous to those used in the Case 1 (i.e., $T_!$ "swallows" any created transactors eliminated by the cycle removal, $M_!$ swallows any messages sent during the cycle, and $T_{\looparrowright}$ and β are updated if the first configuration in the cycle is initial). The first transition of the cycle $c_n^{t_n}$ must be a $[\mathrm{rcv1}]$ rule. If the message received is inconsistent, we remove it from the configuration and add it to the set $T_!$. Otherwise, we add an instance of $[\mathrm{lose}]$ to re-establish the premises of the lemma. The result follows by induction.

Inductive Case 3. In this case, all of the configurations in $c_n^{t_n}$ are inconsistent by assumption, therefore any messages sent during the cycle or any transactors created during the cycle are inconsistent with configuration k_1^{β}. We restore the preconditions for the lemma by removing the cycle and define the sets $T'_{\looparrowright}$, $T'_!$, and $M'_!$ and trace β' as in the previous case. The initial transition of the cycle $c_n^{t_n}$ must be a $[\mathrm{rcv1}]$ rule. If the message received is inconsistent, we remove it from the configuration and add it to the set $T_!$. Otherwise, we add an instance of $[\mathrm{lose}]$ to re-establish the premises of the lemma. The result follows by induction.

Inductive Case 4. If none of the other cases apply, we use $c_n^{t_n}$ to extend β and simply set $\gamma = c_n^{t_n}\beta$. The result follows by induction. $\square$

We are now in a position to prove our main simulation theorem:

THEOREM 1 (SIMULATION WITHOUT NODE FAILURES). *Let k_1 and k_2 be well-formed, resilient, and consistent configurations such that $k_1 \xrightarrow[\tau]{}{}_* k_2$. Then there exists k_2' such that $k_2 \approx k_2'$ and $k_1 \xrightarrow[\smile]{}{}_* k_2'$.*

PROOF. Follows directly from Lemma 5. Define the variables in the premise of the lemma as follows: Let $k_1^{\alpha} = k_1$, $k_2^{\alpha} = k_1^{\beta} = k_2^{\beta} = k_2$. Let α be the unique trace such that $k_1 \xrightarrow[\tau]{\alpha}{}_* k_2$, $\beta = \epsilon$, and $T_{\looparrowright} = T_! = \emptyset$. Given these definitions, all of the premises of Lemma 5 are satisfed trivially, and thus the theorem follows immediately from the lemma. $\square$

The proof of this theorem effectively shows how global reasoning about state inconsistencies can be reduced to local reasoning about the possibility of message loss.

7.3 Universal Checkpointing

The other critical τ-calculus property is *liveness*, i.e., that it is *possible* to reach global checkpoints using the transactor model operational semantics. Of course, not all transactor programs can reach global checkpoints. Indeed, a trivial program with a transactor that sends messages introducing dependencies, but never stabilizes or tries to checkpoint, will eliminate the ability of its dependents to reach checkpoints. We therefore introduce a *Universal Checkpointing Protocol (UCP)* that assumes a set of preconditions that will entail global checkpointing for a set of transactors T. We also prove that under those preconditions, the protocol terminates and therefore, a global checkpoint is reached.

DEFINITION 1 (UCP PRECONDITIONS). *Let D be the set of transactors T and the transitive closure of its dependencies, i.e., all the transactors that elements of T depend on, the transactors that they depend on, and so forth.*

A. All transactors in D need to keep a set of acquaintances, ACQ, in their state since the last checkpoint or time of creation, including the names of:

(1) transactors which have been a target for messages sent.

(2) transactors which have been created.

(3) the parent transactor.

B. All transactors in D need to eventually stabilize and start the Universal Checkpointing Protocol. Also, all transactors in D need to be able to receive ping *messages.*

C. Once the first transactor in D stabilizes, no other transactors in D will programmatically rollback or be caused to rollback by other transactors in D. This assumes previous application-dependent communication that provides this guarantee.

D. There can be no failures while the Universal Checkpointing Protocol is taking place.

DEFINITION 2 (UNIVERSAL CHECKPOINTING PROTOCOL). *When a transactor t in D stabilizes, it:*

I. Pings every transactor in ACQ

II. Checks if it is dependent,

(a) If not, it pings every transactor in ACQ, checkpoints and ends protocol.

(b) If so, it pings every transactor in ACQ and waits for incoming pings*.*

III. On reception of a ping *message, goes back to II.*

Since the UCP protocol only terminates upon Step IIa, successful checkpointing of all transactors in D, protocol termination is sufficient to prove that a global checkpoint has been reached.

We first define a Transactor Dependence Graph (TDG), and prove that incoming edges in this graph (dependencies) can only be created by five specific causality conditions. Then, because of preconditions B..D, eventually the history of all transactors in D will be stable or checkpointed after UCP steps I and II. Furthermore, the stable condition of a transactor will eventually be communicated to all dependent transactors by the UCP protocol. Each transactor's knowledge of the stability of all transactors it depends on, allows it to eventually checkpoint.

DEFINITION 3 (TRANSACTOR DEPENDENCE GRAPH). *Given a transactor configuration, k, we define its transactor dependence graph, $TDG(k)$, as (V, E), where $V = D$, and $\forall\, t_1, t_2 \in V$:*

$$t_1 \overset{h}{\mapsto} t_2 \in E \iff \delta_s(t_1) = h \vee \delta_c(t_1) = h$$

where $k = \_ \mid \theta[t_2 \mapsto \langle \_, \_ ;\ \_, \_ ;\ \delta_s, \delta_c, \_ \rangle]$

In other words, given a transactor configuration, k, its transactor dependence graph ($TDG(k)$) is a labeled directed graph with transactors as nodes and dependencies as labeled edges. The labels represent the last known history information for a given transactor. So, an edge $t_1 \overset{h}{\mapsto} t_2$ represents the fact that t_2 depends on t_1 and t_2 knows h to be the last history value for t_1.

LEMMA 6 (DEPENDENCE CAUSALITY CONDITIONS). *Given a configuration k, if there is an edge in $TDG(k)$ as follows:*

$$t_1 \overset{h}{\mapsto} t_2$$

it is only *because of one of the following conditions:*

i. *transactor t_1 sent a message to t_2;*

ii. *a transactor dependent on t_1 sent a message to t_2;*

iii. *transactor t_1 created t_2;*

iv. *a transactor dependent on t_1 created t_2; or*

v. *transactor t_2 created t_1.*

PROOF. An edge from t_1 to t_2 in the $TDG(k)$ is created only when t_2's δ_c or δ_s are modified to include t_1 in a transition with t_1 or t_2 in focus.

The only transition that adds transactors to t_2's δ_c is $[\mathrm{new}]$, in which t_2's δ_c becomes the creator t's creation and behavioral dependence maps, plus its own history: $\delta_c \oplus \delta_b \oplus [t \mapsto h]$. New edges to t_2 in $TDG(k)$ are created from transactors in this dependence map union. If t_1 is t, t_1 created t_2, which is the condition iii above. If t_1 is in t's δ_c or δ_b, then t is dependent on t_1, which is condition iv.

The only transition that adds transactors to t_2's δ_s is $[\mathrm{set1}]$. It adds transactors in t_2's δ_b. The only transitions that add transactors to t_2's δ_b are $[\mathrm{new}]$, $[\mathrm{rcv1}]$, and $[\mathrm{get}]$. We consider them, one at a time:

$[\mathrm{new}]$ this transition adds $[t' \mapsto \mathbf{H}_0]$, which happens if t_2 created t_1, which is condition v.

$[\mathrm{rcv1}]$ this transition adds transactors in the incoming message dependence map, δ_m. This map, δ_m, is only created in $[\mathrm{send}]$. It contains the sender t_s's creation and behavioral dependence maps, plus its own history: $\delta_c \oplus \delta_b \oplus [t_s \mapsto h]$. New edges to t_2 in $TDG(k)$ are created from transactors in this dependence map union. If t_1 is t_s, t_1 sent a message to t_2, which is the condition i above. If t_1 is in t_s's δ_c or δ_b, then t_s is dependent on t_1, which is condition ii.

$[\mathrm{get}]$: this transition adds no new transactors to t_2's δ_s.

□

THEOREM 2 (UNIVERSAL CHECKPOINTING PROPERTY). *The Universal Checkpointing Protocol (UCP) terminates under UCP preconditions A..D.*

PROOF. When a transactor t becomes stable, all the transactors that depend on it will eventually know through the UCP; either because

1. they are directly dependent on t (conditions i, iii, and v in Lemma 6), and therefore in t's ACQ set by precondition A (Step I will let them know), or
2. they are indirectly dependent on it (conditions ii and iv in Lemma 6), and in this case there is a path of transactors that will eventually ping forward that information (through Step I or II)

The proof consists of two parts: first, we prove the existence of this path of transactors; and second, we prove that the stability information of transactors always gets propagated by the UCP protocol through this path.

Part I. Path Existence.. For every edge e in $TDG(k)$:

$$e : t_1 \overset{h}{\mapsto} t_2$$

there exists a finite path $t_1 = s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n = t_2$ such that $s_{i+1} \in s_i$'s $ACQ, \forall i \in [0..n)$.

By Lemma 6, if there is an edge e in $TDG(k)$, it is only because of conditions i, ii, iii, iv, or v.

Under conditions i, iii, and v, there is a path of length 1, $t_1 = s_0 \rightarrow s_1 = t_2$, because of UCP precondition A.

Under conditions ii and iv, the proof is by induction on the length of the path. We will assume the condition to be true for paths of length $< n$, then:

Condition ii: a transactor t dependent on t_1 sent a message to t_2.

Since t sent a message to t_2, t_2 must be in t's ACQ set, by precondition A1. Therefore, there is a length-one path $t \rightarrow t_2$. Since t is dependent on t_1, there is an edge $t_1 \overset{h}{\mapsto} t$ in $TDG(k)$ and therefore by inductive hypothesis, there is a path $t_1 \rightarrow^* t$ of length $n-1$. Thus, we can create a path $t_1 \rightarrow^* t \rightarrow t_2$ of length n.

Condition iv: a transactor t dependent on t_1 created t_2.

Since t created t_2, t_2 must be in t's ACQ set, by precondition A2. Therefore, there is a length-one path $t \rightarrow t_2$. Since t is dependent on t_1, there is an edge $t_1 \overset{h}{\mapsto} t$ in $TDG(k)$ and therefore by inductive hypothesis, there is a path $t_1 \rightarrow^* t$ of length $n-1$. Thus, we can create a path $t_1 \rightarrow^* t \rightarrow t_2$ of length n.

Part II. UCP Protocol forwards stability information.. Consider any transactor t_1 in D. By precondition B, t_1 will eventually stabilize. According to the $[\mathrm{sta1}]$ and $[\mathrm{sta2}]$ transition rules, its own dependence information, h, after stabilization will be $\langle \mathbf{S}(n), l_h \rangle$ for some n and l_h.

In Step I of the protocol, transactor t_1 sends ping messages to all its acquaintances in the ACQ set. The $[\mathrm{send}]$ rule will put a message m in the network with dependence information carrying the dependence information $[t_1 \mapsto \langle \mathbf{S}(n), l_h \rangle]$.

Consider any acquaintance t_2 in t_1's ACQ set. By precondition B, t_2 is required to be able to receive t_1's ping message. t_2 can be in any of the three following states upon ping message reception:

1. t_2 has not started UCP protocol. (before Step I)
2. t_2 has stabilized but not checkpointed. (end of Step IIb)
3. t_2 has both stabilized and checkpointed. (end of Step IIa)

In case 1, the rule $[\mathrm{rcv1}]$ will sharpen t_2's creation and state dependence maps with the incoming message, thereby making $[t_1 \mapsto \langle \mathbf{S}(n), l_h \rangle]$ be the latest known information about t_1 in t_2.

Because of precondition C, rules $[\mathrm{rol1}]..[\mathrm{rol3}]$ and rules $[\mathrm{rcv2}]..[\mathrm{rcv4}]$ will not apply. Because of precondition D, rules $[\mathrm{fl1}]$ and $[\mathrm{fl2}]$ will not apply.

Furthermore, because of precondition B, t_2 will eventually stabilize and forward t_1's stability information through the acquaintances path (UCP Step I).

In case 2, transactor t_2 will again update its dependence information after $[\mathrm{rcv1}]$. Because of precondition C, rules $[\mathrm{rol1}]..[\mathrm{rol3}]$ and $[\mathrm{rcv2}]..[\mathrm{rcv4}]$ will not apply.

Then, according to UCP Step III, t_2 will try to checkpoint, and independently of success or failure, it will forward t_1's updated stability information to all its acquaintances in ACQ.

In case 3, t_1's dependence information in t_2 must have been stable to succeed checkpointing because of $[\mathrm{sta1}]$'s precondition; therefore, the last round of ping messages in Step IIa must have forwarded t_1's stability to all of t_2's acquaintances.

Since in all three cases, the stable condition of t_1 is properly propagated by t_2 to its ACQ set, and t_1 and t_2 were chosen arbitrarily, we conclude that UCP guarantees that the stable condition of all transactors gets propagated to all their dependents in the TDG. Therefore, all transactors in D eventually successfully reach Step IIa in the UCP protocol, and therefore a global (universal) checkpoint is eventually reached. $\square$

8. DISCUSSION AND FUTURE WORK

In this paper, we have introduced a formal framework for understanding and managing distributed state in the presence of various classes of failures. Internet-scale distributed computing is becoming ever more important as use of Grid mechanisms and web services increases. We believe that in order to develop robust applications in these settings, it is necessary to incorporate state management constructs that are more flexible than traditional transaction mechanisms.

In addition to the failure-free simulation and universal checkpointing properties, there are a number of additional aspects of the τ-calculus that are worthy of further study. For example, one would like to show how certain application properties and topologies allow specialized checkpointing techniques. As a trivial example, consider a transactor t application that reads, but does not update the state of another transactor t'. If t' is initially checkpointed, one can easily show that t' can checkpoint without requiring message exchanges with t. More interestingly, one could define various failure rates and scenarios, and show situations under which configurations are always able to make progress (under reasonable fairness assumptions) despite failures.

Finally, there are a number of interesting directions for further research that build on the ideas developed here, including: modeling transactional *compensation* mechanisms, in which consistency is maintained through *reversal* of actions, rather than rolling back to previous states; modeling *isolation* and *atomicity* in a modular way; studying type systems for statically constraining dependences and exposing various failure modes; developing techniques for optimizing dependence information, and modeling additional classes of failures.

9. ACKNOWLEDGMENTS

We gratefully acknowledge the insightful comments of James Leifer, Carolyn Talcott, Maria-Cristina Marinescu, and the anonymous referees.

10. REFERENCES

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] G. Agha, N. Jamali, and C. Varela. Agent naming and coordination: Actor based models and infrastructures. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, chapter 9, pages 225–246. Springer-Verlag, Mar. 2001.

[3] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[4] M. Berger and K. Honda. The two-phase commitment protocol in an extended pi-calculus. In *Prelim. Proc. EXPRESS '00*, NS-00-2, pages 105–130. BRICS Notes, 2000.

[5] P. A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.

[6] K. P. Birman and R. V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. Wiley-IEEE Computer Society Press, 1994.

[7] L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of System Specification and Computational Structures*, LNCS 1378, pages 140–155. Springer Verlag, 1998.

[8] T. Chothia and D. Duggan. Abstractions for fault-tolerant global computing. *Electronic Notes in Theoretical Computer Science (ENTCS). Foundations of Wide-Area Network Computing (FWAN)*, 66(3), 2002. Elsevier.

[9] J. Field and C. Varela. Towards a programming model for building reliable systems with distributed state. *Electronic Notes in Theoretical Computer Science (ENTCS). First International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA)*, 68(3), 2003. Elsevier.

[10] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 372–385, 1996.

[11] S. Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*. MIT Press, 1996.

[12] S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proc. ECOOP*, volume 707 of *LNCS*, pages 346–360, 1993.

[13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1993.

[14] N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *ACM Trans. Program. Lang. Syst.*, 16(6):1719–1736, 1994.

[15] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8-3:323–364, June 1977.

[16] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Proceedings of Supercomputing'95*, 1995.

[17] B. Liskov. Distributed programming in Argus. *Communications of the Association of Computing Machinery*, 31(3):300–312, 1988.

[18] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I-II. *Information and Computation*, 100(1):1–77, 1992.

[19] Object Management Group. CORBA services: Common object services specification version 2. Technical report, Object Management Group, June 1997. http://www.omg.org/corba/.

[20] A. Spector, R. Pausch, and G. Bruell. Camelot: A flexible, distributed transaction processing system. In *Proc. IEEE Computer Society International Conf.*, pages 432–437, San Francisco, March 1988. IEEE Computer Society Press.

[21] Sun Microsystems, Inc. Java Remote Method Invocation specification, Rev. 1.8. ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf, 2002.

[22] C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3), 1998.

[23] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[24] C. Varela and G. Agha. A Hierarchical Model for Coordination of Concurrent Activities. In P. Ciancarini and A. Wolf, editors, *Third International Conference on Coordination Languages and Models (COORDINATION '99)*, LNCS 1594, pages 166–182, Berlin, April 1999. Springer-Verlag. http://osl.cs.uiuc.edu/Papers/Coordination99.ps.

[25] J. Waldo. JINI Architecture Overview, 1998. Work in progress. http://www.javasoft.com/products/jini/.

[26] World Wide Web Consortium. Web services activity statement. http://www.w3.org/2002/ws/, 2002.

[27] X/Open Company Limited. Distributed transaction processing: The XA specification. X/Open Company Limited, 1991. X/Open CAE Specification XO/CAE/91/300.