Optimistic Concurrency Semantics for Transactions in Coordination Languages

Suresh Jagannathan and Jan Vitek

Department of Computer Sciences
Purdue University
West Lafayette, IN 47906
{suresh,jv}@cs.purdue.edu

Abstract. There has been significant recent interest in exploring the role of coordination languages as middleware for distributed systems. These languages provide operations that allow processes to dynamically and atomically access and manipulate collections of shared data. The need to impose discipline on the manner in which these operations occur becomes paramount if we wish to reason about correctness in the presence of increased program complexity. Transactions provide strong serialization guarantees that allow us to reason about programs in terms of higher-level units of abstraction rather than lower-level data structures. In this paper, we explore the role of an optimistic transactional facility for a Linda-like coordination language. We provide a semantics for a transactional coordination calculus and state a soundness result for this semantics. Our use of an optimistic concurrency protocol distinguishes this work from previous efforts such as Javaspaces, and enables scalable, deadlock-free implementations.

1 Introduction

A transaction defines a locus of computation that adheres to well-known safety and failure properties. To ensure these properties hold, the semantics of a transactional facility must guarantee that the effects of a transaction are not observable until a commit occurs; when a commit does occur, all effects are propagated instantaneously to the enclosing parent transaction; once a transaction commits, the global state records its effects, and forgets the association between the effect and the transaction; and, all data accesses performed by a transaction are performed serially with respect to other concurrently executing transactions. These four traits correspond to isolation, atomicity, durability, and consistency properties respectively in classical transaction models.

There have been several attempts which explore the integration of transactional semantics within a coordination language framework. Most notably, JavaSpaces [17] and TSpaces [26] combine Linda-like semantics augmented with transactional features into Java. A formal treatment of JavaSpaces, along with

several important extensions, is given by Busi and Zavattaro [10, 11]. The JavaSpaces implementation is based on a *pessimistic* concurrency control semantics: locks are acquired when an entry is accessed, preventing other transactions from witnessing that value until the owning transaction commits. An entry can be physically removed from the shared space only if the removing action is part of the same transaction as any uncommitted reads to that object. Writes to the space become visible to other transactions only when their initiator commits. Although relatively simple to describe, a pessimistic treatment of transactions for coordination languages has two notable disadvantages:

1. *Deadlock*: Since shared data are locked by reads, two transactions each of whom read distinct entries may deadlock depending upon the order in which the locks are acquired. Moreover, it is not possible to acquire locks prior to executing a transaction since determining whether an object will be read depends upon the transaction's dynamic control-flow. The ability to select entries based on pattern-matching further complicates the issue of deadlock detection and avoidance.
2. *Scalability*: Pessimistic concurrency control negatively impacts scalability of distributed implementations as read operations require the acquisition of global locks and thus increase synchronization between distributed nodes. Non-blocking operations, such as Linda's RDP and INP (which test for the presence of a tuple) further complicate the implementation of pessimistic concurrency control protocols as they require that every *potential* entry that may match the operation's template argument be locked for the duration of the transaction [10] to preserve desired serializability invariants.

In this paper, we explore an alternative treatment of transactions within a tuple space coordination framework that addresses these issues. We define a semantics for a transactional variant of Linda [18] based on the synchronous π-calculus [23]. This language can be viewed as the computational core of systems such as JavaSpaces (without some features such as leases and notification [17]) in which effects on the shared data space are controlled via a transactional mechanism based on an *optimistic* concurrency protocol. In this protocol, every transaction conceptually operates over its own local copy of the space and performs actions restricted to this copy. When a transaction is to commit, the state of the local copy is compared to the current state of the global space. If serializability invariants have been violated, the transaction aborts and its local copy is discarded. Otherwise, the changes can be propagated to the global space.

Our work is distinguished from previous efforts in three main respects: (*a*) our transactional semantics is based on an *optimistic* concurrency model that eliminates issues of deadlock, and obviates scalability limitations which arise in more pessimistic lock-based transactional schemes; (*b*) our treatment of non-blocking operations that test for the presence or absence of tuples, allow us to reason about the validity of these operations using transaction-local, rather than global, judgments; and (*c*) the semantics naturally allows transactions to be arbitrar-

ily nested. Taken together, these features suggest that expressive, scalable, and deadlock-free transactional implementations for coordination languages are feasible.

2 Coordination Languages and Transactions

We are interested in defining robust, scalable concurrent programs that use a Linda-style coordination framework to mediate access to shared data. Classical coordination models such as Linda allow atomic access and modification of content-addressable data. However, explicit support for treating a locus of computation and the collection of data the computation accesses as a single atomic unit is not provided. An atomic transaction is a control abstraction that provides such functionality. The transactional semantics of concurrent actions in the context of a coordination model ensures serializability and atomicity properties of operations that read, remove, write, and test for the presence of tuples in a global data space.

2.1 JavaSpaces

Sun Microsystem's JavaSpaces is a well-known attempt at integrating a Linda-like coordination language enriched with transactional support into a general purpose programming language. Although expressive, the JavaSpaces programming model is somewhat unintuitive and difficult to implement. We consider its main features here and refer interested readers to the specification [17] and related research papers [7, 10, 11, 8, 25] for further clarification. In particular, we do not consider leasing and event notification which are treated in detail by Busi and Zavattaro in [10, 11].

In JavaSpaces a shared data space is an object which supports at least the following operations: `write(), read(), take(), readIfExists()` and `takeIfExists()`. The first operation deposits an entry in the shared space. The operations `read( template)` and `write( template)` scan the shared space for an entry matching `template`, reading or removing a matching tuple, respectively, if one exists and blocking otherwise. Although the match operation in JavaSpaces is user-defined, we restrict ourselves in this paper to traditional value-based structural pattern matching familiar to pure Linda systems. If the template is the distinguished `null` value, any entry in the data space may match. `readIfExists()` and `takeIfExists()` are non-blocking equivalents of `read()` and `take()` respectively (similar to Linda's `rdp` and `inp`). If no value matching the template is currently in the shared space, these operations return `null`. Busi and Zavarroto have discussed in detail the implications of the ability to test for the presence of a value [10, 11]. We revisit these operations in later sections.

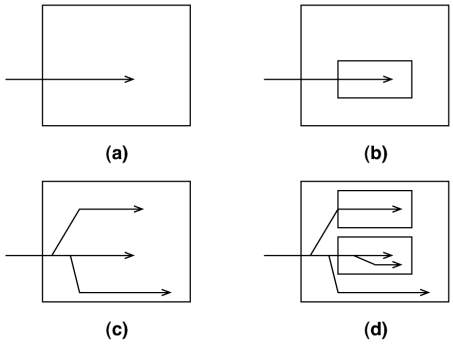

Fig. 1. Threads and transactions may be interleaved in various ways: (a) plain, (b) nested, (c) multi-threaded, (d) multi-threaded and nested.

2.2 Transactions

In this paper, we consider a multi-threaded nested transactional model. When transactions are *nested* [24], each top-level transaction is divided into a number of child transactions; each of which can also have nested transactions. Nested transactions commit from the bottom up, and child transactions must always commit before their parent. A transaction abort at one level does not necessarily affect a transaction in progress at a higher level. The updates of committed transactions at intermediate levels are visible only within the scope of their immediate predecessors. Support for nested transactions is an important feature of our semantics. A nested transaction defines a locus of computation whose effects are bounded by its parent. Thus, the effects of operations on shared data can be controlled by defining an appropriate transaction nesting structure, leading to improved modularity and scalability. Programs are more modular because the effects of a nested transaction are localized to its parent. Programs are scalable because effects are not propagated immediately as the nested transaction commits; instead, changes are aggregated and made globally visible only when the parent transaction commits.

In a multi-threaded transaction model, each transaction can have multiple concurrent threads operating on the same view of the shared state. In a nested transaction model, multiple threads can be executing in parent and child transaction concurrently. Fig. 1 summarizes the different interactions that arise between transactions and threads.

A transactional facility can be modeled by a simple API consisting of the operation `start()` and `commit()`. We assume an implicit transactional context so that `start()` initiates a new transaction. If the current thread is already running within a transaction, the new transaction is nested in the current one, otherwise the new transaction becomes a top-level transaction. The `commit()` operation

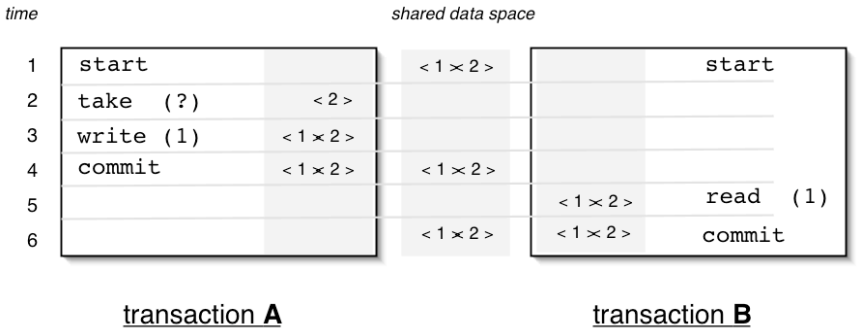

Fig. 2. Serialized execution of transaction `A` and `B`. The shared data space is only updated upon commit. Each transaction has its own view of the data space. The shared space starts out as { ⟨1⟩, ⟨2⟩ }. At time 2, transaction `A` performs a `take(?)` which removes entry ⟨1⟩ from the space. Thus from `A`'s point of view the shared space contains a single entry (⟨2⟩). A commit reconciles a transaction's local view with the state of the shared space.

attempts to finalize the changes made by the current transaction, propagating its results to the parent if one exists.

2.3 Motivating examples

To illustrate some salient issues related to the incorporation of transactional semantics into a coordination language, consider the actions defined in Fig. 2. Transactions `A` and `B` perform read and write actions on a global data space which initially has two singleton tuples ⟨1⟩ and ⟨2⟩. Transaction `A` takes ⟨1⟩ and then writes it back. Once `A` is done, transaction `B` performs a non-destructive read of ⟨1⟩ and commits. This is a valid interleaving as all actions of `A` precede the actions of `B`.

Figs. 3 and 4 illustrates possible interleavings if the actions performed by `A` and `B` are enclosed within transactions under different transactional semantics. In Fig. 3 transaction `B` reads ⟨1⟩ after it was taken by `A`. In a *pessimistic* model such as the one used by JavaSpaces [10], this interleaving is not valid as the `take()` performed by `A` locks ⟨1⟩ and prevents `B` from performing a read. In an *optimistic* semantics, transaction `B` can proceed and, in fact, both transactions can commit successfully.

Fig. 4 illustrates a case in which, under an *optimistic* semantics, transaction `B` has to abort. In this example, `A` reads ⟨1⟩ but writes back another copy of ⟨2⟩, while `B` still reads ⟨1⟩. Of course when `B` gets to the point of committing, it finds that the global space does not have tuple ⟨1⟩ and thus is forced to abort.

Fig. 5 illustrates a more serious problem with JavaSpaces. We use `inp` as a non-blocking take (*i.e.* `takeIfExists()`). Assume the global data space is initially empty. Transaction `A` checks if tuple $\langle 2 \rangle$ is present, and will take if it is, while transaction `B` attempts to take $\langle 1 \rangle$. Both operations fail as the space contains

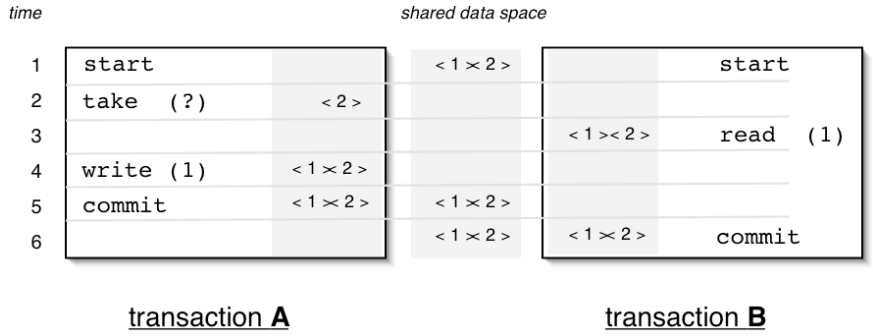

Fig. 3. Valid optimistic interleaving. In a pessimistic semantics transaction `B` would have to block until `A` commits, *i.e.* it would behave as illustrated in Fig. 2.

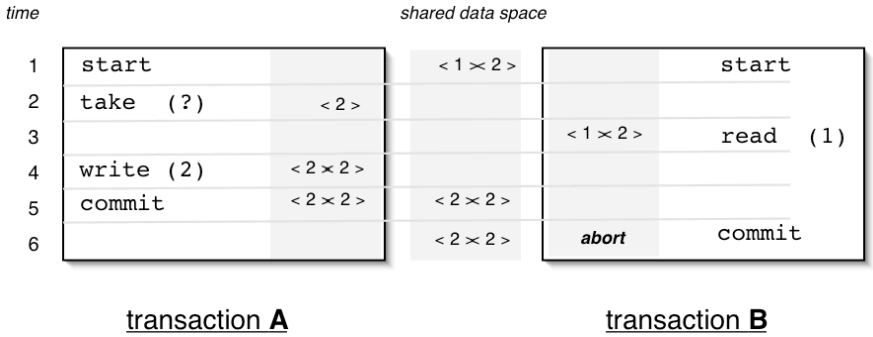

Fig. 4. Valid optimistic interleaving. Transaction `B` fails to commit due to a conflict in the global data space (entry $\langle 1 \rangle$ is absent).

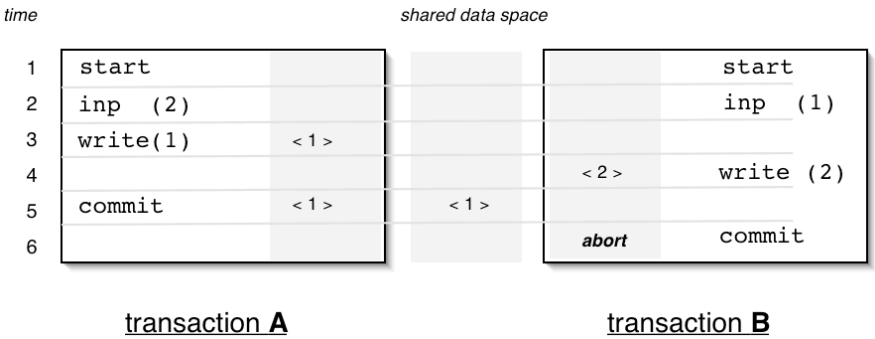

Fig. 5. Valid optimistic interleaving. Transaction `B` abort due to a conflict in the state of the shared space ($\langle 1 \rangle$ is present). A pessimistic semantics would deadlock.

no matching value. `A` proceeds to write $\langle 1 \rangle$ and `B` to write $\langle 2 \rangle$, after which both transactions commit. In an *optimistic* semantics transaction `B` would abort because at time 6 the contents of the global space is $\{\langle 1 \rangle\}$ while transaction `B` expect $\langle 1 \rangle$ *not* to be in the space (since it has checked this at time 2). In Sun's JavaSpace *implementation*, both transactions are allowed to commit successfully. This behavior is incorrect as noted by Busi and Zavattaro [10]. In a correct *pessimistic semantics* (*i.e.* [10]) this problem can be redressed by prohibiting other transactions from depositing any matching writes to the tuple-space once `A` performs its test. However, in this example, such a semantics would lock any transaction from depositing a single element tuple, and would thus prevent `B` from committing since it subsequently performs a `write` on $\langle 2 \rangle$.

Thus, the example of Fig. 5 would deadlock in a pessimistic semantics because each transaction is trying to acquire a lock held by the other transaction. In general, a correct implementation of pessimistic concurrency control may entail locking large portions of the shared data space. Consider for instance the action `readIfExists( null)` which, in JavaSpaces, matches *any* entry in the data space. If the data space is empty when the action is evaluated, the current transaction will lock the entire space and no other transaction will be able to write to the space until this one commits if serializability invariants are to be preserved.

Transaction protocols for coordination languages are further complicated when nested transactions are considered. The semantics of nested transactions permit child transactions to see the effects of their parents, even before the parent commits. Using pessimistic concurrency control, the rules governing when tuple operations become visible, and when read and write locks are released must be strengthened. Fig. 6 describes a valid execution sequence in which transaction `A` defines a nested transaction `C`. Tuples written by `C` are unavailable to `B` until `A` commits, although they are available to `A` once `C` commits. Tuples written by `A` are available to `C` even before `A` commits. This figure reveals that fine-grained pessimistic concurrency protocols require sophisticated lock management schemes to allow nested transactions to inherit locks on tuples owned by their ancestors. In constrast, an optimistic concurrency protocol maintains multiple local logs and directly incorporates notions of visibility into the log structure. When a nested transaction attempts to read a tuple, writes performed by its parent that match the tuple pattern are visible to the nested transaction. To ensure that writes performed in a nested transaction are propagated correctly, an optimistic concurrency mechanism need only guarantee that nesting of logs reflect transaction nesting. No special lock management protocol is required.

3 A transactional coordination calculus

The transactional Linda calculus is based on the synchronous π calculus [1, 3, 21] which provides a small and elegant computational core. The main departure

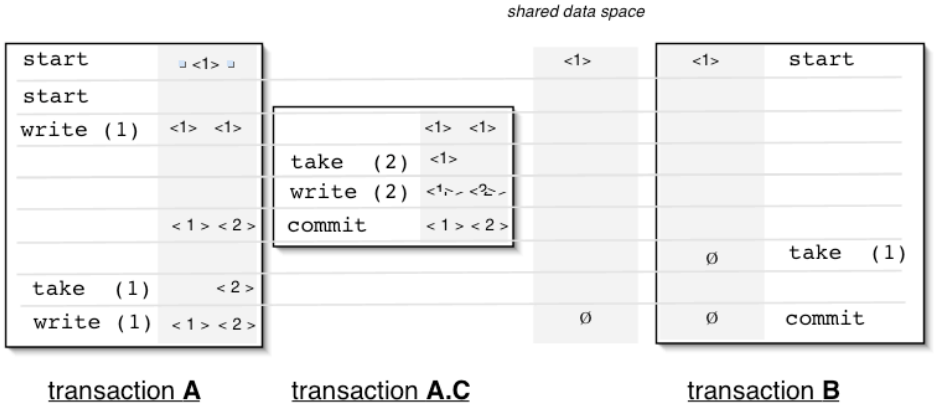

Fig. 6. Actions performed by a nested transaction are not visible to the parent until the inner transaction commits; the effects of a nested transaction are not visible to other transactions until its parent commits.

from the π calculus is that rather than communicating by means of channel-based message passing, processes exchange tuples through a common shared space. Embeddings of Linda in process calculi have been explored in previous work [5, 16]. We shall focus only on essential features from the point of view of the transactional behavior of the calculus.

3.1 Syntax

The syntax of the calculus is summarized in Fig. 7. We take an infinite set of names ranged over by meta-variable `x`. Tuples are sequences of formal and actual entries ranged over by meta-variable `v`. The empty tuple is denoted by $\langle\rangle$. The expression `x . v` denotes the tuple resulting from concatenating an actual name `x` with tuple `v`, while $\mathtt{x}^{?}\,.\,\mathtt{v}$ denotes the concatenation of a formal variable `x` with `v`. Meta-variable `t` ranges over transaction identifiers, and ℓ over sequences of transaction identifiers.

The syntactic category of processes, ranged over by `P` and `Q`, includes the empty process **0** and parallel composition of processes `P | Q`. Synchronous output, written `v. P`, deposits the tuple denoted by `v` in the shared data space and proceeds to execute `P`. The input operation `(v). P` evaluates the template `v` against the shared space; if a matching value is found, the tuple is taken from the shared space and the process continues as `P` with the formals of `v` replaced by actuals from the tuple; the operation blocks if no matching value can be found. Guarded replication, written `(v)!P`, performs an input of `v` and, if a match is found, proceeds as `P` with (in parallel) a copy of the original process `(v)!P`. The test operation, `(v)?P ; Q` is a non-blocking test for presence which tries to find a tuple matching the template `v`, and if one is found, proceeds to evaluate `P` with the tuple; if no matching tuple is found in the space `Q` is evaluated. The restriction operator

Syntax:

$\mathtt{P} ::= \mathtt{0} \mid \mathtt{P}|\mathtt{Q} \mid (\nu\,\mathtt{x})\,\mathtt{P} \mid \mathbf{trans}\,\mathtt{P} \mid \mathtt{t}[\mathtt{P}] \mid \mathbf{commit} \mid (\mathtt{v}).\mathtt{P} \mid \mathtt{v}.\mathtt{P} \mid (\mathtt{v})?\mathtt{P};\mathtt{Q} \mid (\mathtt{v})!\mathtt{P}$

$\mathtt{v} ::= \langle\rangle \mid \mathtt{x}.\mathtt{v} \mid \mathtt{x}^{?}.\mathtt{v} \qquad \tau ::= \mathtt{v} \mid {}^{\mathtt{v}}\!/_{\mathtt{v}} \mid \neg\mathtt{v} \mid \mathbf{com} \mid \mathbf{srt} \qquad \ell ::= \langle\rangle \mid \ell.\mathtt{t}$

$\mathcal{E} ::= \langle\rangle \mid \mathcal{E}.\ell : \rho \qquad \rho ::= \langle\rangle \mid \rho.\mathtt{v} \mid \rho.\neg\mathtt{v} \mid \rho.\mathtt{v}^{?}$

$$(\mathtt{v}).\mathtt{P} \xrightarrow{{}^{\mathtt{v}'}\!/_{\mathtt{v}}}_{\langle\rangle} {}^{\mathtt{v}'}\!/_{\mathtt{v}}\,\mathtt{P} \quad \text{(L-INP)}$$

$$(\mathtt{v})?\mathtt{P};\mathtt{Q} \xrightarrow{\mathtt{v}}_{\langle\rangle} {}^{\mathtt{v}'}\!/_{\mathtt{v}}\,\mathtt{P} \quad \text{(L-POS)}$$

$$(\mathtt{v})?\mathtt{P};\mathtt{Q} \xrightarrow{\neg\mathtt{v}}_{\langle\rangle} \mathtt{Q} \quad \text{(L-NEG)}$$

$$(\mathtt{v})!\mathtt{P} \xrightarrow{{}^{\mathtt{v}'}\!/_{\mathtt{v}}}_{\langle\rangle} {}^{\mathtt{v}'}\!/_{\mathtt{v}}\,\mathtt{P} \mid (\mathtt{v})!\mathtt{P} \quad \text{(L-LOOP)}$$

$$\mathtt{v}.\mathtt{P} \xrightarrow{\mathtt{v}}_{\langle\rangle} \mathtt{P} \quad \text{(L-OUTP)}$$

$$\mathbf{trans}\,\mathtt{P} \xrightarrow{\mathbf{srt}}_{\mathtt{t}} \mathtt{t}[\mathtt{P}] \quad \text{(L-START)}$$

$$\mathtt{t}[\mathbf{commit} \mid \mathtt{P}] \xrightarrow{\mathbf{com}}_{\mathtt{t}} \mathbf{0} \quad \text{(L-COMM)}$$

$$\frac{\mathtt{P} \xrightarrow{\tau}_{\ell} \mathtt{P}'}{\mathtt{t}[\mathtt{P}] \xrightarrow{\tau}_{\ell.\mathtt{t}} \mathtt{t}[\mathtt{P}']} \quad \text{(L-TRANS)}$$

$$\frac{\mathtt{P} \xrightarrow{{}^{\mathtt{v}'}\!/_{\mathtt{v}}}_{\ell} \mathtt{P}' \quad \mathtt{v}', \mathcal{E}' = \mathtt{take}(\mathcal{E}, \ell) \quad \mathtt{v}' \le \mathtt{v}}{\mathtt{P}\ \mathcal{E} \overset{\ell}{\Longrightarrow} \mathtt{P}'\ \mathcal{E}'} \quad \text{(G-INP)}$$

$$\frac{\mathtt{P} \xrightarrow{\mathtt{v}}_{\ell} \mathtt{P}' \quad \mathcal{E}' = \mathtt{put}(\mathcal{E}, \ell, \mathtt{v})}{\mathtt{P}\ \mathcal{E} \overset{\ell}{\Longrightarrow} \mathtt{P}'\ \mathcal{E}'} \quad \text{(G-OUTP)}$$

$$\frac{\mathtt{P} \xrightarrow{\neg\mathtt{v}}_{\ell} \mathtt{P}' \quad \mathcal{E}' = \mathtt{neg}(\mathcal{E}, \ell, \mathtt{v})}{\mathtt{P}\ \mathcal{E} \overset{\ell}{\Longrightarrow} \mathtt{P}'\ \mathcal{E}'} \quad \text{(G-NEG)}$$

$$\frac{\mathtt{P} \xrightarrow{\mathbf{srt}}_{\ell} \mathtt{P}' \quad \mathcal{E}' = \mathtt{start}(\mathcal{E}, \ell)}{\mathtt{P}\ \mathcal{E} \overset{\ell}{\Longrightarrow} \mathtt{P}'\ \mathcal{E}'} \quad \text{(G-START)}$$

$$\frac{\mathtt{P} \xrightarrow{\mathbf{com}}_{\ell} \mathtt{P}' \quad \mathcal{E}' = \mathtt{commit}(\mathcal{E}, \ell)}{\mathtt{P}\ \mathcal{E} \overset{\ell}{\Longrightarrow} \mathtt{P}'\ \mathcal{E}'} \quad \text{(G-COMM)}$$

$$\frac{\mathtt{P}\ \mathcal{E} \equiv \mathtt{P}'\ \mathcal{E} \quad \mathtt{P}'\ \mathcal{E} \overset{\ell}{\Longrightarrow} \mathtt{Q}'\ \mathcal{E}' \quad \mathtt{Q}' \equiv \mathtt{Q}}{\mathtt{P}\ \mathcal{E} \overset{\ell}{\Longrightarrow} \mathtt{Q}\ \mathcal{E}'} \quad \text{(G-CONG)}$$

Evaluation Contexts:

$$E[\bullet] \mid E[\bullet] \mid P \mid \mathtt{t}[E[\bullet]]$$

$$\frac{\mathtt{P} \xrightarrow{\tau}_{\ell} \mathtt{P}'}{E[\mathtt{P}] \xrightarrow{\tau}_{\ell} E[\mathtt{P}']} \qquad \frac{\mathtt{P} \xrightarrow{\tau}_{\ell} \mathtt{P}' \quad \mathtt{x} \notin fn(\tau)}{(\nu\,\mathtt{x})\,\mathtt{P} \xrightarrow{\tau}_{\ell} (\nu\,\mathtt{x})\,\mathtt{P}'}$$

Structural Congruence:

$\mathtt{P} \mid \mathtt{Q} \equiv \mathtt{Q} \mid \mathtt{P} \qquad (\nu\,\mathtt{x})\ (\nu\,\mathtt{y})\ \mathtt{P} \equiv (\nu\,\mathtt{y})\ (\nu\,\mathtt{x})\ \mathtt{P}$

$\mathtt{P} \mid (\nu\,\mathtt{x})\ \mathtt{Q} \equiv (\nu\,\mathtt{x})\ \mathtt{P} \mid \mathtt{Q} \quad \textit{if } \mathtt{x} \notin fn(\mathtt{P})$

$\mathtt{t}[(\nu\,\mathtt{x})\ \mathtt{P}] \equiv (\nu\,\mathtt{x})\ \mathtt{t}[\mathtt{P}]$

$(\nu\,\mathtt{x})\ \mathtt{P}\ \ \mathcal{E} \equiv \mathtt{P}'\ \ \mathcal{E} \qquad \textit{if } \mathtt{x} \notin fn(\mathcal{E}) \wedge \mathtt{P} \equiv \mathtt{P}'$

Free Names:

$fn(\mathbf{0}) = \{\} \qquad fn(\mathtt{P} \mid \mathtt{Q}) = fn(\mathtt{P}) \cup fn(\mathtt{Q}) \qquad fn((\nu\,\mathtt{x})\ \mathtt{P}) = fn(\mathtt{P}) - x \qquad fn(\mathtt{t}[\mathtt{P}]) = fn(\mathtt{P})$

$fn(\mathbf{commit}) = \{\} \qquad fn((\mathtt{v}).\mathtt{P}) = fn(\mathtt{P}) \cup fn(\mathtt{v}) - bn(\mathtt{v}) \qquad fn(\mathtt{v}.\mathtt{P}) = fn(\mathtt{P}) \cup fn(\mathtt{v}) - bn(\mathtt{v})$

$fn(\mathbf{trans}\,\mathtt{P}) = fn(\mathtt{P}) \qquad fn((\mathtt{v})?\mathtt{P};) = fn(\mathtt{P}) \cup fn(\mathtt{v}) - bn(\mathtt{v}) \qquad fn((\mathtt{v})!\mathtt{P}) = fn(\mathtt{P}) \cup fn(\mathtt{v}) - bn(\mathtt{v})$

$fn(\mathtt{x}.\mathtt{v}) = \{x\} \cup fn(\mathtt{v}) \qquad fn(\mathtt{x}^{?}.\mathtt{v}) = fn(\mathtt{v}) \qquad fn(\langle\rangle) = \{\}$

$bn(\mathtt{x}.\mathtt{v}) = bn(\mathtt{v}) \qquad bn(\mathtt{x}^{?}.\mathtt{v}) = \{x\} \cup bn(\mathtt{v}) \qquad bn(\langle\rangle) = \{\}$

Matching:

$\langle\rangle \le \langle\rangle \qquad \mathtt{x}.\mathtt{v} \le \mathtt{x}.\mathtt{v}' \ \textit{ if } \mathtt{v} \le \mathtt{v}' \qquad \mathtt{x}^{?}.\mathtt{v} \le \mathtt{x}.\mathtt{v}' \ \textit{ if } \mathtt{v} \le \mathtt{v}'$

Fig. 7. Syntax and Semantics

$(\nu\,\mathtt{x})\ \mathtt{P}$ generates a fresh name $\mathtt{x}$; the calculus is lexically scoped, so $\mathtt{x}$ is visible only in process P. Finally **trans** $\mathtt{P}$ and **commit** are used to, respectively, start and terminate a new transaction. The transaction **trans** $\mathtt{P}$ will execute the process $\mathtt{P}$ until a **commit** is evaluated, at which point all changes effected by the transaction will become visible to all other transactions. If a conflict is detected the transaction will not be able to commit and the evaluation of the thread will remain stuck. Note also that all other threads within the same transaction and nested transaction are terminated upon a commit. The calculus does not provide an explicit **abort** operation; we model aborted transactions as stuck terms on **commit** transitions.

3.2 Semantics

The semantics of the calculus is shown in Figs. 7 and 8. The semantics is stratified so that the local reduction relation $\mathtt{P} \xrightarrow{\tau}_{\ell} \mathtt{Q}$ defines that process $\mathtt{P}$ can reduce to $\mathtt{Q}$ in one step. The transition is labeled by an action τ and a transaction ℓ. The global relation $\mathtt{P}\ \mathcal{E} \stackrel{\ell}{\Longrightarrow} \mathtt{Q}\ \mathcal{E}'$ defines the behavior of a program $\mathtt{P}$ in an environment $\mathcal{E}$. We assume congruence of environments under element reordering.

In the local reduction relation the tuple space remains implicit. Each reduction step is labeled by one of the following actions: $\mathtt{v}, {}^{\mathtt{v}'}\!/_{\mathtt{v}}, \neg\mathtt{v}, \mathbf{srt}$, and **com**, indicating a write of $\mathtt{v}$, a take of $\mathtt{v}'$ matching $\mathtt{v}$, a test for absence of $\mathtt{v}$, the start of a transaction and a commit, respectively. Furthermore transitions are labeled by the issuing transaction name, ℓ, composed of a sequence of transaction identifiers; the different identifiers correspond to the levels of nesting, thus $\mathtt{t}.\mathtt{t}'.\mathtt{t}'''$ denotes an action performed by transaction $\mathtt{t}'''$ nested in transactions $\mathtt{t}'$ and $\mathtt{t}$.

Starting a transaction with **trans** $\mathtt{P}$ creates a process $\mathtt{P}$ running within some transaction $\mathtt{t}$, and is denoted $\mathtt{t}[\,\mathtt{P}\,]$. We assume that each new transaction identifier $\mathtt{t}$ is chosen to be distinct. Transactions can be nested, for instance the

$$
{}^{\mathtt{x}.\mathtt{v}}\!/_{\mathtt{x}.\mathtt{v}'}\ \mathtt{P} = {}^{\mathtt{v}}\!/_{\mathtt{v}'}\ \mathtt{P} \qquad {}^{\mathtt{x}.\mathtt{v}}\!/_{\mathtt{y}^?.\mathtt{v}'}\ \mathtt{P} = {}^{\mathtt{v}}\!/_{\mathtt{v}'}\ {}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{P} \qquad {}^{\langle\,\rangle}\!/_{\langle\,\rangle}\ \mathtt{P} = \mathtt{P}
$$

$$
{}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathbf{0} = \mathbf{0} \qquad {}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{P} \mid \mathtt{Q} = {}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{P} \mid {}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{Q} \qquad {}^{\mathtt{x}}\!/_{\mathtt{y}}\ (\nu\,\mathtt{x})\ \mathtt{P} = (\nu\,\mathtt{x})\ \mathtt{P}
$$

$$
{}^{\mathtt{x}}\!/_{\mathtt{y}}\ (\nu\,\mathtt{x}')\ \mathtt{P} = (\nu\,\mathtt{x}')\ {}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{P} \qquad {}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{t}[\,\mathtt{P}\,] = \mathtt{t}[\,{}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{P}\,] \qquad {}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathbf{commit} = \mathbf{commit}
$$

$$
{}^{\mathtt{x}}\!/_{\mathtt{y}}\ (\mathtt{v}).\,\mathtt{P} = ({}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{v}).\,{}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{P} \quad \textit{if } x \notin \mathit{fn}(\mathtt{v}) \qquad {}^{\mathtt{x}}\!/_{\mathtt{y}}\ (\mathtt{v}).\,\mathtt{P} = ({}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{v}).\,\mathtt{P} \quad \textit{if } x \in \mathit{fn}(\mathtt{v})
$$

$$
{}^{\mathtt{x}}\!/_{\mathtt{y}}\ (\mathtt{v})!\,\mathtt{P} = ({}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{v})!\,{}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{P} \quad \textit{if } x \notin \mathit{fn}(\mathtt{v}) \qquad {}^{\mathtt{x}}\!/_{\mathtt{y}}\ (\mathtt{v})!\,\mathtt{P} = ({}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{v})!\,\mathtt{P} \quad \textit{if } x \in \mathit{fn}(\mathtt{v})
$$

$$
{}^{\mathtt{x}}\!/_{\mathtt{y}}\ (\mathtt{v})?\,\mathtt{P}\,;\,\mathtt{Q} = ({}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{v})?\,{}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{P}\,;\,{}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{Q} \quad \textit{if } x \notin \mathit{fn}(\mathtt{v}) \qquad {}^{\mathtt{x}}\!/_{\mathtt{y}}\ (\mathtt{v})?\,\mathtt{P}\,;\,\mathtt{Q} = ({}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{v})?\,\mathtt{P}\,; \quad \textit{if } x \in \mathit{fn}(\mathtt{v})
$$

$$
{}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{v}.\,\mathtt{P} = ({}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{v}).\,{}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{P} \quad \textit{if } x \notin \mathit{fn}(\mathtt{v}) \qquad {}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{x}.\mathtt{v} = y.{}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{v} \qquad {}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{x}^?.\mathtt{v} = \mathtt{x}^?.\mathtt{v}
$$

$$
{}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{z}^?.\mathtt{v} = \mathtt{z}.{}^{\mathtt{x}}\!/_{\mathtt{y}}\ \mathtt{v} \qquad {}^{\mathtt{x}}\!/_{\mathtt{y}}\ \langle\,\rangle = \langle\,\rangle
$$

Fig. 8. Substitutions.

expression $\mathtt{t}[\,\mathtt{t}'[\,\mathtt{P}\,]\mid\mathtt{Q}\,]$ denotes two parallel processes P and Q such that Q is running within a top level transaction t and a process P in a transaction $\mathtt{t}'$ nested within t. For example, the expression **trans** (**trans** $\mathtt{v}.\mathbf{0}$) denotes a process that will spawn two transactions, the second nested within the first. Then the expression will reduce in one step to the inactive process:

$$\mathtt{t}[\ \mathtt{t}'[\mathtt{v}.\mathbf{0}]\ ] \quad \xrightarrow{\mathtt{v}}_{\mathtt{t}\,.\,\mathtt{t}'} \quad \mathtt{t}[\ \mathtt{t}'[\mathbf{0}]\ ]$$

Notice that in this configuration, the tuple v was output from transaction $\mathtt{t}\,.\,\mathtt{t}'$ and since there is no commit for that transaction that value will never become available to other processes.

The global reduction relation $\stackrel{\ell}{\Longrightarrow}$ manages the shared data space and defines the semantics of the transactional facility. The semantics does not fix a specific implementation or a particular transactional model as it is parameterized by an environment $\mathcal{E}$ which contains the tuple space and five operations over environments that implement the actual transactional model. Although we provide definitions for these operations that capture the essential traits of an optimistic concurrency protocol, other specifications that define alternative implementations of the protocol, or which express different transactional semantics (such as pessimistic concurrency) can be developed without modifying the global reduction relation.

The rule (G-INP) defines the behavior of a destructive read over the tuple space. Given an environment and a transaction, `take` returns a value available to that transaction; if that value matches the requested template, the transition proceeds with an updated environment. (G-OUTP) is similar in that it relies on the `put` operation to record that a transaction ℓ has output tuple v. (G-NEG) applies in case there is no tuple matching template v available to transaction ℓ. When this occurs, the `neg` operation records this fact in the environment. (G-START) sets up the environment for a new transaction. (G-COM) attempts to commit a transaction. Finally, (G-CONG) allows reduction up to structural congruence of processes. Top level ν-binders can be erased to allow names to flow into the environment.

3.3 Optimistic transactional facility

The implementation of an optimistic transactional model is shown in Figs. 9 and 10. In this scheme, the shared tuple space is represented by an environment $\mathcal{E} = \ell_1 : \rho_1 \ldots \ell_n : \rho_n$ which contains per-transaction logs $\rho_1 \ldots \rho_n$. Each of these logs is a sequence comprised of events, $\mathtt{v}, \mathtt{v}^?, \neg\mathtt{v}$; these events denote an output of v, a removal of a tuple $\mathtt{v}^?$ and an absence of a match for template $\neg\mathtt{v}$, respectively.

The operations on the environment are `take`, `neg`, `put`, `commit` and `start`. Whenever a new transaction is created $\mathtt{start}(\mathcal{E}, \ell)$ is used to extend the environment

$$\frac{\mathtt{v} \in \mathtt{visible}(\mathcal{E}, \ell) \quad \mathcal{E}' = \mathcal{E} \,.\, (\ell : \rho \,.\, \mathtt{v}^?)}{\mathtt{v}, \mathcal{E}' = \mathtt{take}(\mathcal{E}, \ell)} \qquad \frac{\mathtt{match}(\mathtt{visible}(\mathcal{E}, \ell), \mathtt{v}) = \{\} \quad \mathcal{E}' = \mathcal{E} \,.\, (\ell : \rho \,.\, \neg \mathtt{v})}{\mathcal{E}' = \mathtt{neg}(\mathcal{E}, \ell, \mathtt{v})}$$

$$\frac{\mathcal{E}' = \mathcal{E} \,.\, (\ell : \rho \,.\, \mathtt{v})}{\mathcal{E}' = \mathtt{put}(\mathcal{E}, \ell, \mathtt{v})} \qquad \frac{\mathcal{E}' = \mathtt{reflect}(\mathcal{E}'', \rho, \ell)}{\mathcal{E}' = \mathtt{commit}(\mathcal{E}'' \,.\, (\ell \,.\, \mathtt{t} : \rho), \ell)}$$

$$\mathcal{E} \,.\, (\ell : \langle\,\rangle) = \mathtt{start}(\mathcal{E}, \ell)$$

Fig. 9. Transactional semantics.

$$\mathcal{E} = \mathtt{reflect}(\mathcal{E}, \langle\,\rangle, \ell) \qquad \frac{\mathcal{E}' = \mathtt{reflect}(\mathcal{E} \,.\, (\ell : \rho' \,.\, \mathtt{v}), \rho, \ell)}{\mathcal{E}' = \mathtt{reflect}(\mathcal{E} \,.\, (\ell : \rho'), \mathtt{v} \,.\, \rho, \ell)}$$

$$\frac{\mathtt{v} \in \mathtt{visible}(\mathcal{E}, \ell) \quad \mathcal{E}' = \mathtt{reflect}(\mathcal{E} \,.\, (\ell : \rho' \,.\, \mathtt{v}^?), \rho, \ell)}{\mathcal{E}' = \mathtt{reflect}(\mathcal{E} \,.\, (\ell : \rho'), \mathtt{v}^? \,.\, \rho, \ell)} \qquad \frac{\mathtt{match}(\mathtt{visible}(\mathcal{E}, \ell), \mathtt{v}) = \{\} \quad \mathcal{E}' = \mathtt{reflect}(\mathcal{E} \,.\, (\ell : \rho' \,.\, \neg \mathtt{v}), \rho, \ell)}{\mathcal{E}' = \mathtt{reflect}(\mathcal{E} \,.\, (\ell : \rho'), \neg \mathtt{v} \,.\, \rho, \ell)}$$

$$\mathtt{visible}(\mathcal{E}, \ell) = \mathtt{find}(\mathtt{merge}(\mathcal{E}, \ell)) \qquad \mathtt{match}(\mathcal{V}, \mathtt{v}) = \{\, \mathtt{v}' \mid \mathtt{v}' \in \mathcal{V} \,\wedge\, \mathtt{v} \leq \mathtt{v}' \}$$

$$\begin{array}{l} \mathtt{find}(\rho \,.\, \mathtt{v}) = \mathtt{find}(\rho \cup \mathtt{v}) \\ \mathtt{find}(\rho \,.\, \mathtt{v}^?) = \mathtt{find}(\rho - \mathtt{v}) \\ \mathtt{find}(\rho \,.\, \neg \mathtt{v}) = \mathtt{find}(\rho) \end{array} \qquad \begin{array}{l} \mathtt{merge}(\mathcal{E}, \langle\,\rangle) = \langle\,\rangle \\ \mathtt{merge}(\mathcal{E} \,.\, (\ell \,.\, \mathtt{t} : \rho), \ell \,.\, \mathtt{t}) = \mathtt{merge}(\mathcal{E}, \ell) \,.\, \rho \end{array}$$

Fig. 10. Auxiliary functions.

with an empty log for transaction ℓ. The operation $\mathtt{put}(\mathcal{E}, \ell, \mathtt{v})$ extends the log of transaction ℓ with a tuple $\mathtt{v}$. The operation $\mathtt{take}(\mathcal{E}, \ell)$ returns an arbitrary tuple visible by transaction ℓ and records in the log that the tuple has been removed. The operation $\mathtt{neg}(\mathcal{E}, \ell, \mathtt{v})$ records the fact no tuple matching template $\mathtt{v}$ is visible to the transaction ℓ. Finally, operation $\mathtt{commit}(\mathcal{E}, \ell)$ attempts to commit the change performed by transaction ℓ to the log of the parent transaction. Commit operates by replaying the changes performed by transaction ℓ. The commit fails if the environment for ℓ's parent is not in the expected state.

3.4 Soundness

Proving the correctness of the semantics requires that we show our treatment of transactions preserves desired isolation and atomicity properties. To do so, we first define a notion of a *well-defined* state. Intuitively, a state is well-defined if the contents of the logs associated with each transaction are such that no conflicts would arise. One way to manifest this idea is to compare the state of a transaction's parent with its child at the point the child attempts to commit.

If the parent reflects commits from other transactions that violate invariants recorded in the log of the committing child, the child state is not considered well-defined. We formalize this intuition thus:

Definition 1. *$\mathcal{E}$ is well-defined if for any transaction ℓ such that $\mathcal{E} \equiv \mathcal{E}' \,.\, (\ell : \rho)$ and $\ell = \ell' \,.\, \mathtt{t}$, the function $\mathtt{reflect}(\mathcal{E}', \ell', \rho)$ is defined.*

Thus, the log of a child transaction is well-defined with respect to the parent if actions observed by the child are consistent with the actions seen by the parent. Satisfiability of this condition implies, that after a child commits, `write` and `take` operations performed by the child must be visible in the parent, and tuples that were observed to be absent by the child must still be so in the parent.

Definition 2. *A trace $tr(R) = \mathtt{P}_0\,\mathcal{E}_0 \stackrel{\ell_0}{\Longrightarrow} \ldots \stackrel{\ell_n}{\Longrightarrow} \mathtt{P}_n\,\mathcal{E}_n$ is* serial *iff $\forall i, j, k$ such that $0 \leq i \leq j \leq k \leq n$, and $\ell_i = \ell_k$, $l_i \lhd l_j$ (read "l_i is a prefix of l_j).*

A serial trace is one in which for all pairs of reduction steps with the same transaction label ℓ, all actions that occur between these two steps are taken on behalf of that transaction or its children. Given a notion of a serial trace, we can define a soundness theorem that captures our desired notion of serializability:

Theorem 1. *Let R be a sequence of reductions $\mathtt{P}_0\,\mathcal{E}_0 \stackrel{\ell_0}{\Longrightarrow} \ldots \stackrel{\ell_n}{\Longrightarrow} \mathtt{P}_{n_1}\,\mathcal{E}_{n+1}$. If $\mathcal{E}_{n+1}$ is well-defined, then there exists a sequence $R' = \mathtt{P}_0\,\mathcal{E}_0 \stackrel{\ell_0}{\Longrightarrow} \ldots \stackrel{\ell'_n}{\Longrightarrow} \mathtt{P}_{n_1}\,\mathcal{E}_{n+1}$ such that R' is serial.*

The proof of this theorem appeals to notions of permutability on global actions. Informally, two actions α_1 and α_2 executed in transactions ℓ_1 and ℓ_2 are permutable if they have no data or control-dependency with each other. For example, a `take` operation performed by a transaction ℓ has a data dependency with any `write` operation performed by any of ℓ's parents that matches the read tuple. Simlarly, a `write` action logged in any parent transaction of ℓ has a data dependency with any $\neg\mathtt{v}$ action recorded in ℓ. This means that any valid serializable permutation cannot move a `write` action in a parent above a child action that successfully tested for the absence of the tuple being written.

4 Related Work and Conclusions

The Linda coordination model [18, 13] uses generative communication on anonymous structured data to facilitate interactions among concurrent programs. Its simplicity and generality make it a fertile vehicle in which to explore and formalize various concurrency abstractions [8, 16, 6, 12]. However, Linda does not directly support atomic operations on aggregate shared data. To alleviate this

drawback, systems such as JavaSpaces [17] or TSpaces [26] allow operations on tuple-spaces to be encapsulated within transactions.

There is a large body of work that explores the formal specification of various flavors of transactions [22, 15, 19]. Most closely related to our work is that of Bussi, Gorrieri and Zavattoro [7] and Busi and Zavattoro [10] who formalize the semantics of JavaSpaces and discuss the semantics of important extensions such as leasing [9]. However, their work is presented in the context of a pessimistic concurrency control model. Our contribution is a formal characterization of transactions for Linda based on optimistic concurrency that provides scalable and deadlock-free execution.

Other related efforts include the work of Black *et. al.* [2] and Choithia and Duggan [14]. The former presents a theory of transactions that specify atomicity, isolation and durability properties in the form of an equivalence relation on processes. Choithia and Duggan present the pik-calculus and pike-calculus, extension of the pi-calculus that supports various abstractions for distributed transactions and optimistic concurrency. Their work is related to other efforts [10, 4] that encode transaction-style semantics into the pi-calculus and its variants. Our work is distinguished from these efforts in that it provides a simple operational characterization and proof of correctness of transactions that can be used to explore different trade-offs when designing a transaction facility for incorporation into a language. Haines *et.al.* [20] describe a composable transaction facility in ML that supports persistence, undoability, locking and threads. Their abstractions are very modular and first-class, although their implementation does not rely on optimistic concurrency mechanisms to handle commits.

References

1. Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96*, volume 1119 of *LNCS*, pages 147–162. Springer-Verlag, Berlin, 1996.
2. Andrew Black, Vincent Cremet, Rachid Guerraoui, and Martin Odersky. An Equational Theory for Transactions. Technical Report CSE 03-007, Department of Computer Science, OGI School of Science and Engineering, 2003.
3. Gérard Boudol. Asynchrony and the π-calculus (Note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.
4. R. Bruni, C. Laneve, and U. Montanari. Orchestrating Transactions in the Join Calculus. In *13^{th} International Conference on Concurrency Theory*, 2002.
5. Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. A process algebraic view of Linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, February 1998.
6. Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. A process algebraic view of linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.
7. Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. On the Semantics of JavaSpaces. In *Formal Methods for Open Object-Based Distributed Systems IV*, volume 177. Kluwer, 2000.

8. Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. Process calculi for coordination: From Linda to JavaSpaces. *Lecture Notes in Computer Science*, 1816, 2000.
9. Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. Temporary Data in Shared Dataspace Coordination Languages. In *FOSSACS'01*, pages 121–136. Springer-Verlag, 2001.
10. Nadia Busi and Gianluigi Zavattaro. On the serializability of transactions in JavaSpaces. In *Proc. of International Workshop on Concurrency and Coordination (CONCOORD'01)*. Electronic Notes in Theoretical Computer Science 54, Elsevier, 2001.
11. Nadia Busi and Gianluigi Zavattaro. On the serializability of transactions in shared dataspaces with temporary data. In *Proc. of ACM Symposium on Applied Computing (SAC'02)*, pages 359–366. ACM Press, 2002.
12. N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *LNCS*, pages 66–76. Springer-Verlag, Berlin, 1995.
13. Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.
14. Tom Chothia and Dominic Duggan. Abstractions for Fault-Tolerant Computing. Technical Report 2003-3, Department of Computer Science, Stevens Institute of Technology, 2003.
15. Panos Chrysanthis and Krithi Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.
16. R. DeNicola and R. Pugliese. A Process Algebra based on Linda. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061 of *Lecture Notes in Computer Science*, pages 160–178. Springer-Verlag, Berlin, 1996.
17. Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, Reading, MA, USA, 1999.
18. David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
19. Jim Gray and Andreas Reuter. *Transaction Processing*. Morgan-Kaufmann, 1993.
20. Nicolas Haines, Darrell Kindred, Gregory Morrisett, Scott Nettles, and Jeannette Wing. Composing First-Class Transactions. *ACM Transactions on Programming Languages and Systems*, 16(6):1719–1736, 1994.
21. Kohei Honda and Mario Tokoro. On Asynchronous Communication Semantics. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing. LNCS 612*, pages 21–51, 1992.
22. Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan-Kaufmann, 1994.
23. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100, September 1992.
24. J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Massachusetts, 1985.
25. Jaco van de Pol and Miguel Valero Espada. Formal specification of JavaSpaces^{TM} Architecture using μCRL. *Lecture Notes in Computer Science*, 2315, 2002.
26. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.