A Transactional Object Calculus

Suresh Jagannathan Jan Vitek Adam Welc Antony Hosking

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

This article is an extended version of a paper presented in the 2004 European Symposium on Programming entitled, *A Semantic Framework for Designer Transactions*[24].

Abstract. A transaction defines a locus of computation that satisfies important concurrency and failure properties. These so-called ACID properties provide strong serialization guarantees that allow us to reason about concurrent and distributed programs in terms of higher-level units of computation (*e.g.*, transactions) rather than lower-level data structures (*e.g.*, mutual-exclusion locks). This paper presents a framework for specifying the semantics of a transactional facility integrated within a host programming language. The TFJ calculus, an object calculus derived from Featherweight Java, supports nested and multi-threaded transactions. We give a semantics to TFJ that is parameterized by the definition of the transactional mechanism that permits the study of different transaction models. We give two instantiations: one that defines transactions in terms of a versioning-based optimistic concurrency model, and the other which specifies transactions in terms of a pessimistic two-phase locking protocol, and present soundness and serializability properties for our semantics.

1 Introduction

The integration of transactional facilities into programming languages has been driven by applications ranging from middleware infrastructure for enterprise applications [1] to runtime support for optimistic concurrency. The concept of transactions is well-known in database systems; the main challenge in associating transactions with programming control structures comes from the mismatch between the concurrency model of the programming language and the concurrency model of the transactional facility. Issues related to the interaction commit mechanisms found in transactional models with locking, visibility, and update semantics in programming languages can be subtle and complex. Existing technologies enforce little or no relationship between these models, so that programmers can neither rely on transactions for complete isolation among concurrent threads, nor use concurrent threading to more conveniently program transaction logic. To make matters worse, different systems exhibit subtly (or vastly) different observable behavior with respect to failure and isolation properties. While the differences are often mandated by pragmatic concerns, they make it hard to reason about and compare different systems. Furthermore correctness of an implementation often remains an article of faith.

As a first step towards addressing some of these concerns, we propose a semantic framework in which different transactional mechanisms can be studied and compared formally. Requirements for such a framework are that it be sufficiently expressive to allow the specification of core transactional features, that it provide a way to validate the correctness of the semantics, and that it support features found in realistic programs. We are interested in the impact of design choices on observable behavior (*e.g.*, aborts, deadlocks, livelocks) and on implementation performance (*e.g.*, space and time overhead). Our long-term goal is to leverage this framework to aid in the definition of program analyzes and optimization techniques for transactional languages.

This paper introduces TFJ, or *Transactional Featherweight Java*, an object calculus with syntactic support for transactions. The operational semantics of TFJ is given in terms of a stratified set of rewrite rules parameterized over the meaning of the transactional constructs. This allows us to define a variety of transactional semantics within the same core language. In this paper, we study *nested and multi-threaded* transactions with two different concurrency control models (two-phase locking and versioning). We have tried to retain the flavor and economy of Featherweight Java [17], while adding features such as concurrency and state needed to model transactions. The primary contribution of this paper is a formal characterization and proof of correctness of different transactional models when incorporated into the core language. While there has been significant previous work on devising formal notation and specifications [7, 20] describing transactional properties, we are unaware of other efforts that use operational semantics to study the interplay of concurrency and serializability among different transactional models. Instead, we explore the structure of program traces induced by the semantics, and prove a simpler permutation lemma on these traces that enforce a localized notion of serializability derived from control and dependencies among transaction operations. The proof of serializability over arbitrary traces follows naturally from the composition of permutable sub-traces.

Correctness of the transactional semantics is given in terms of the traditional ACID properties [15] described below. To prove correctness, we show that all local operations performed by concurrent transactions can be *serialized* (*i.e.*, from the perspective of some transaction T all data accesses performed by T are performed serially with respect to other transactions). The use of a stratified semantics allows us to avoid proving this serialization theorem for different concurrency control models. Instead, we explore the structure of program traces induced by the semantics, and prove a simpler permutation lemma on these traces that enforce a localized notion of serializability derived from control and dependencies among transaction operations.

Overview. Section 2 gives an informal introduction to atomic transaction and the ACID properties. A taxonomy of the different concurrency control protocols with different observable behavior and performance characteristics appears in Section 3. The Transactional Featherweight Java calculus is introduced in Section 4. Section 5 and Section 6 give, respectively, versioning and strict two phase locking semantics to the calculus. The soundness results are given in Section 7. Observations about the calculus is given in Section 8. Related work is discussed in Section 9 and Section 10 concludes.

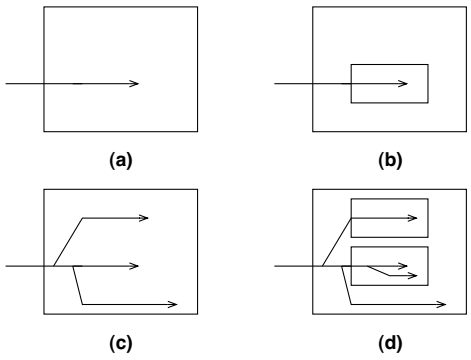

Fig. 1. Threads and transactions may be interleaved in various ways: (a) plain, (b) nested, (c) multi-threaded, (d) multi-threaded and nested.

2 Atomic Transactions

Atomic transactions [15] are a control abstraction that arose in the database community to delimit logical units of data processing. They permit concurrent data access among transactions, with operations on shared data interleaved to the extent that they do not violate the so-called "ACID" transaction semantics, ensuring:

1. *Atomicity*: A transaction is a sequence of operations that is performed *atomically*, either in its entirety or else not at all. If it completes successfully, it *commits*. Otherwise, it *aborts* and has no effects.
2. *Isolation*: A transaction should not make its effects visible to other transactions until it commits. The effect is that concurrent transactions are *serializable* in that they appear to occur one-at-a-time.
3. *Consistency*: A correct execution of a transaction takes shared data from one consistent state (as defined by the application semantics) to another provided that transactions execute atomically and in isolation.
4. *Durability*: Once it has committed, the effects of a transaction survive subsequent system failures.

Transactions may be *nested* [22], with each top-level transaction divided into a number of child transactions. Because transactions commit from the bottom up, 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. Effects performed by a parent transaction are always visible to the child.

Concurrency control ensures the consistency property, usually via protocols that guarantee transaction serializability. Executions are consistent if they execute in a way that is serializable. Concurrency protocols are classified as either pessimistic or optimistic. The former detect violations of serializability property early but usually incur significant runtime and space overheads, the latter are less expensive but detect serializability violations

as late as the commit time. *Two-phase locking* (2PL) [13] is a protocol that requires all lock operations on behalf of a transaction to precede the *first* unlock operation in the transaction, resulting in two computation phases: a growing phase as locks are acquired and a shrinking phase as they are released. In *strict* 2PL a transaction does not release any of its locks until after it commits or aborts. 2PL can be extended to handle nested transactions as follows [22]: a transaction may acquire a lock if all owner are ancestors of the transaction and when a transaction commits, all its locks are handed back to the parent or released if the transaction is top-level. Optimistic protocols are based on a simple idea. Every read or write is remembered in a private *journal* (or log) instead of being directly applied to the global data. The transaction proceeds uninterrupted until commit time, but before it is allowed to commit, the controller performs conflict detection. If the serializability property has been violated then the transaction aborts and all the data in its private log is abandoned. There are various ways of performing conflict detection, *e.g.* comparing initial (remembered at the first access) values of data used by a transaction with values of the same data at commit time. Extending optimistic schemes to incorporate nested transactions is reasonably straightforward and involves modifying the commit procedure. Every transaction keeps a separate journal and at commit time the updates performed by a nested transaction are propagated from its private journal to the journal of its immediate parent. The updates are propagated to the global data set only after a successful commit of the top-level transaction.

3 A Taxonomy of Protocols

For a given transactional model, such as the nested and multi-threaded one studied here, there exist a range of concurrency control protocols with different observable behavior and performance characteristics. While we do not purport to present an exhaustive list, we list some known interesting points in the spectrum.

- **Precise (pessimistic)**: Block a transaction T iff it will definitely have a conflict with a transaction T'. This is in general undecidable without an oracle.
- **Strict 2PL**: Block a transaction T if a *potential* conflict is detected with a transaction T'. Under strict 2PL locks on objects maybe acquired before objects are referenced or modified with new values.
- **2PL**: Similar to strict 2PL except that lock acquisition is less aggressive. Locks are acquired only when necessary to ensure atomicity properties. Note that an apparent conflict such as two writes to the same variable may not be a real conflict, *e.g.* both writes affect the same value to the variable and no intervening read observes the change.
- **Byzantine**: An optimistic scheme in which any transaction can be aborted at any time, this even if the transaction will not have a conflict. All optimistic transactions can lead to livelock.
- **Fail-fast**: Abort a transaction T as soon as a potential conflict has been detected with some transaction T'. Thus, every read and write checks whether serialization invariants on the target object have been violated, and aborts if so.
- **Precise (optimistic)**: Abort a transaction T if, and only if, a conflict has been detected with some transaction T' when T is about to commit. This protocol delays

aborts, thus allowing the possibility that a transaction performing a write action which would have led to an abort in a fail-fast scheme is allowed to commit because subsequent actions mask the effect of the write.

The tradeoffs between these protocols range from programming model issues (which kinds of failures should the programmer be exposed to, how quickly should failures be detected, and how should deadlocks and livelocks be resolved) to implementation performance. For example, using memory pages as the unit of granularity for detecting conflicts may be efficient, but may lead to unpredictable or frequent aborts. From a programmer's point of view this is akin to byzantine failures in distributed systems. While a more detailed and formal investigation of these alternatives remains as an open problem, we pick two points in the design space –precise optimism (which we refer to as a *versioning* instantiation) and strict 2PL (which we refer to as a *locking* instantiation) – and give their semantics in the following sections.

4 The Transactional Featherweight Java Calculus

Transactional Featherweight Java (TFJ) is a new object calculus inspired by the work of Igarashi *et al.* [17]. TFJ includes threads and the imperative constructs needed to model transactions. We focus on a simplified variant of TFJ, that is dynamically typed and in which all classes directly inherit the distinguished `Object` class[1]. To introduce the syntax and semantics of TFJ, we start with a simple example.

Consider the classes given in Fig. 2. Class `Updater` encapsulates an update to an object. Class `Runner` is designed to perform an action within a new thread. Class `Transactor` performs two operations within a transaction. In more detail, class `Updater` has two fields, `n` and `v` and an `update` method which assigns `v` to `n`'s `val` field. `Runner` has a `run` method which starts a new thread and invokes a method on its `r` field within that thread. `Transactor` has a `pick` method which is used to evaluate two expressions in a non-deterministic order; non-determinism is achieved since the order in which arguments are evaluated in a method call is unspecified. It also has a `run` method which starts a new transaction and invokes `update` on field `u` and `run()` on field `r`. The keyword **onacid** marks the beginning of a transaction and **commit** ends the current transaction. All objects have an `init` method which assigns values to their fields and returns `this`.

Fig. 3 gives a TFJ code fragment making use of the above class definitions. Variable `n` is bound to a new object of some class `Number` (whose only feature of interest is that it must have a `val` field; we further assume the existence of classes `One`, `Two`, and `Three` that define specific numbers). `Noop` is an initialized `Runner` object with an uninteresting `run` method. Objects `l1` and `l2` are transactors which will be used to initiate nested transaction (`l2` within `l1`). Two runner objects will be used to create threads $\mathtt{t}_1$ and $\mathtt{t}_2$.

Evaluating the program of Fig. 3 will result in the creation of two threads ($\mathtt{t}_1$ and $\mathtt{t}_2$) and two new transactions ($\mathtt{l}_1$ and $\mathtt{l}_2$). Thread $\mathtt{t}_1$ executes solely within transaction $\mathtt{l}_1$,

[1] Even though types and inheritance are central features of object-oriented languages they are orthogonal to issues studied in this paper. The interaction of concurrency and inheritance is well-studied [2, 21], and no additional novel problems are posed by considering object inheritance in the context of transactions.

while $\mathtt{t}_2$ starts executing in $\mathtt{l}_1$, before starting transaction $\mathtt{l}_2$. We assume that there is a default top-level transaction, $\mathtt{l}_0$ and primordial thread $\mathtt{t}_0$. Fig. 4 shows the structure of this computation. The threads in a parent transaction can execute concurrently with threads in nested transactions. A design choice in TFJ is that all threads must join (via a commit) for the entire transaction to commit. Alternatives, such as interrupting threads that are not at a commit point when another thread in the same transaction is ready to commit, or silently committing changes while the thread is running are either programmer unfriendly or counter to the spirit of transactions.

The state in this program is defined by the instance of class `Number` that is threaded through the transactions and handed down to `Updater`s for modification. Each invocation of `update()` performs a read and a write of the `val`. One valid interleaving of the operations is, for example:

$$[\mathtt{n} := \mathtt{One}()]_{\mathtt{l}_1} \;\rightarrow\; [\mathtt{n}]_{\mathtt{l}_1} \;\rightarrow\; [\mathtt{n} := \mathtt{Two}()]_{\mathtt{l}_2} \;\rightarrow\; [\mathtt{n}]_{\mathtt{l}_2} \;\rightarrow\; [\mathtt{n} := \mathtt{Three}()]_{\mathtt{l}_0}$$

This is correct because all of the changes performed by $\mathtt{l}_1$ occur before changes of transactions $\mathtt{l}_2$ and $\mathtt{l}_1$. An example of an invalid interleaving of these operations is:

$$[\mathtt{n} := \mathtt{One}()]_{\mathtt{l}_1} \;\rightarrow\; [\mathtt{n} := \mathtt{Two}()]_{\mathtt{l}_2} \;\rightarrow\; [\mathtt{n}]_{\mathtt{l}_1} \;\rightarrow\; [\mathtt{n}]_{\mathtt{l}_2} \;\rightarrow\; [\mathtt{n} := \mathtt{Three}()]_{\mathtt{l}_0}$$

In this schedule, serializability is broken because $\mathtt{l}_1$ reads the value of `n.val` that was changed by $\mathtt{l}_2$. Thus from $\mathtt{l}_1$'s viewpoint the global state is `n.val` = `Two`(). Most concurrency control protocols will flag this as a conflict and abort $\mathtt{l}_1$. We note that in this particular case the conflict is benign as $\mathtt{l}_1$ discards the value it read and thus the state of the system is not affected by it reading a stale value.

```
class Updater {
   n, v;
   init( n, v) { this.n := n; this.v := v; this; }
   update() { this.n.val:= this.v; }
}

class Runner {
   r;
   init( r) { this.r := r; this; }
   run() { spawn this.r.run(); }
}

class Transactor {
   u, r;
   init( r, u) { this.u := u; this.r := r; this; }
   run() {
      onacid;
      this.u.update(); this.r.run(); this.u.n.val;
      commit;
   }
}
```

Fig. 2. Example Class definitions.

```
n := new Number();
s1 := new Transactor.init( Noop, new Updater().init( n, new One()));
r1 := new Runner().init( s1);
s2 := new Transactor.init( r1, new Updater().init( n, new Two()));
new Runner().init( s2).run();
n.val := new Three()
```

Fig. 3. A TFJ code fragment using definition of Fig. 2.

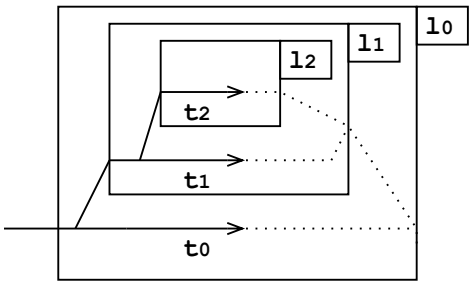

Fig. 4. Threads and transactions in Fig. 3. Threads $\mathtt{t}_0$, $\mathtt{t}_1$, $\mathtt{t}_2$, execute, respectively, in transactions $\mathtt{l}_0$, $\mathtt{l}_1$, $\mathtt{l}_2$. The transactions are nested, such that $\mathtt{l}_0$ is the outermost transaction. Transaction commits are implicit thread joins.

4.1 Syntax

The syntax of TFJ is given in Fig. 5. We take metavariables `L` to range over class declarations, `C`, `D` to range over classes, `M` to range over methods, and `f` and `x` to range over fields and parameters, respectively. We also use `P` for process terms, and `e` for expressions. We assume that the keyword `this` is included in the set of variables, but that `this` is never used as the name of an argument to a method. We use `v` to range over basic expressions, and `r` and `u` to range over references. The only values in TFJ are references created by `newC`() operations, and the special reference `null`. Field selection, method call, and assignment are subject to evaluation if their constituent elements are references.

We use over-bar to represent a finite ordered sequence, for instance, $\overline{\mathtt{f}}$ represents $\mathtt{f}_1\,\mathtt{f}_2\,\ldots\,\mathtt{f}_n$. The term $\overline{\mathtt{l}}\mathtt{l}$ denotes the extension of the sequence $\overline{\mathtt{l}}$ with a single element `l`, and $\overline{\mathtt{l}}\,.\,\overline{\mathtt{l}'}$ for sequence concatenation. We write $\overline{\mathtt{l}} \lhd \overline{\mathtt{l}}'$ if $\overline{\mathtt{l}}$ is a prefix of $\overline{\mathtt{l}}'$.

A process term `P` can be either the empty process `0`, the parallel composition of processes $P \mid P$ or a thread `t` running expression `e`, denoted `t`[`e`]. The thread label `t` is distinct for every thread.

The calculus has a call-by-value semantics. The expression $\mathtt{C}(\overline{\mathtt{r}})\downarrow_{\mathtt{i}}^{\mathtt{r}'}$ yields an object identical to $\mathtt{C}(\overline{\mathtt{r}})$ except in the ith field which is set to $\mathtt{r}'$. The `null` metavariable is used to represent an unbound reference. By default all objects are null initialized (*i.e.* $\mathtt{C}(\overline{\mathtt{null}})$).

Since TFJ has by-value semantics for invocation, sequencing can be encoded as a sequence of method invocations. For readability, we sometimes write "$(\mathtt{e}_1;\mathtt{e}_2)$" in examples to

indicate sequencing of expressions $\mathtt{e}_1$ and $\mathtt{e}_2$. The value of a sequence is always the value of the last expression.

An *expression* `e` can be either a variable `x`, the `this` pseudo variable, a field access `e.f`, a method invocation $\mathtt{e.m}(\overline{\mathtt{e}})$, an object construction `newC`(), a thread creation `spawn e`, an `onacid` command or a `commit`. The latter three operations are unique to TFJ. The expression `spawn e` creates a new thread of control to evaluate `e`. The evaluation of `e` takes place in the same environment as the thread executing `spawn e`. A new transaction is started by executing `onacid`. The dynamic context of `onacid` is delimited by `commit`. Effects performed within the context of `onacid` are not visible outside the transaction until a `commit` occurs. Transactions may be nested. When the `commit` of an inner transaction occurs, its effects are propagated to its parent. Threads may be spawned within the context of a transaction. The local state of the transaction is visible to all threads which execute within it. Transactions may also execute concurrently. For example, in `spawn e`, `e` may be an expression that includes `onacid` and `commit`; the transaction created by `onacid` executes concurrently with the thread executing the `spawn` operation.

Note that the language does not provide an explicit `abort` operation. Transactions may abort *implicitly* because serialization invariants are violated. Our semantics expresses implicit aborts both in the definition of `commit` and in the treatment of read and write operations that would otherwise expose violations of necessary serializability invariants. Implicit aborts are tantamount to stuck states.

4.2 Reduction

The dynamic semantics of our language shown in Figs. 5 and 6 is given by a two-level set of rewrite rules. The computational core of the language is defined by a reduction relation of the form $\mathcal{E}\ \mathtt{e} \xrightarrow{\alpha} \mathcal{E}'\ \mathtt{e}'$. Here $\mathcal{E}$ is a sequence of *transaction environments*, `e` is an expression and the action label α determines which reduction was picked. Each transaction environment consists of a *transaction label*, and a binding environment that maps references to objects ($\mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{r}})$), Action labels for the computational core are selected from the set $\{rd, wr, xt\}$, respectively denoting read, write and extend. In addition to specifying the action on whose behalf a particular reduction is taken, we also specify the action's effects; for example, we write $wr\ vv'$to denote an action with label wr which has effect on locations v and v'. A read action effects the location being read, a write action has an effect on both the location being written and the location whose value it reads, and an extend operation has an effect on the newly created location.

A second reduction relation $\stackrel{\alpha}{\Longrightarrow}_{\mathtt{t}}$ defines operations over the entire program and has the form $\Gamma\ P \stackrel{\alpha}{\Longrightarrow}_{\mathtt{t}} \Gamma'\ P'$ where Γ is a program state composed of a sequence of thread environments $\overline{\mathtt{t}, \mathcal{E}}$ where each $\mathtt{t}, \mathcal{E}$ pair represent the association of a thread to its transaction environments. The action label α can be one of the computational core labels or one of $\{sp, ac, co, ki\}$ for, respectively, spawn, onacid, commit, and kill. As with core actions, the actions corresponding to these labels have an effect on the global state; these effects are given in brackets. Thus, a `spawn` action has the effect of creating a new thread with label `t`; an `onacid` action creates a new transaction with label `l`; a `commit` operation has an effect on the current transaction; and, a `kill` action has an effect on the current thread.

Syntax:

$$\begin{array}{l}
\mathtt{P} ::= \mathtt{0} \mid \mathtt{P}|\mathtt{P} \mid \mathtt{t}[\,\mathtt{e}\,] \\
\mathtt{L} ::= \mathtt{class\ C}\ \{\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \} \\
\mathtt{M} ::= \mathtt{m}(\overline{\mathtt{x}})\,\{\,\mathtt{e};\,\} \\
\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{e.f} := \mathtt{e} \mid \\
\qquad \mathtt{new\ C}() \mid \mathtt{spawn\ e} \mid \mathtt{onacid} \mid \mathtt{commit} \mid \mathtt{null} \\
\mathtt{v} ::= \mathtt{r} \mid \mathtt{v.f} \mid \mathtt{v.m}(\overline{\mathtt{v}}) \mid \mathtt{v.f} := \mathtt{v}
\end{array}$$

Field look-up:

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C}\ \{\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \}}{\mathit{fields}(\mathtt{C}) = (\overline{\mathtt{f}})}$$

Method body look-up:

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C}\ \{\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \} \qquad \mathtt{m}(\overline{\mathtt{x}})\,\{\,\mathtt{e};\,\} \in \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m},\ \mathtt{C}) = (\overline{\mathtt{x}},\ \mathtt{e})}$$

Local Computation:

$$\frac{\mathcal{E}', \mathtt{C}(\overline{\mathtt{u}}) = \mathit{read}(\mathtt{r}, \mathcal{E}) \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{f}})}{\mathcal{E}\ \mathtt{r.f_i} \xrightarrow{rd\ \mathtt{r}} \mathcal{E}'\ \mathtt{u}_i} \qquad \text{(R-Field)}$$

$$\frac{\mathcal{E}', \mathtt{C}(\overline{\mathtt{r}}) = \mathit{read}(\mathtt{v}, \mathcal{E}) \quad \mathcal{E}'' = \mathit{write}(\mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}})\downarrow_{\mathtt{i}}^{\mathtt{r}'}, \mathcal{E}')}{\mathcal{E}\ \mathtt{r.f_i} := \mathtt{r}' \xrightarrow{wr\ \mathtt{r}\mathtt{r}'} \mathcal{E}''\ \mathtt{r}'} \qquad \text{(R-Assign)}$$

$$\frac{\mathcal{E}', \mathtt{C}(\overline{\mathtt{r}}) = \mathit{read}(\mathtt{r}, \mathcal{E}) \quad \mathit{mbody}(\mathtt{m},\ \mathtt{C}_0) = (\overline{\mathtt{x}},\ \mathtt{e})}{\mathcal{E}\ \mathtt{r.m}(\overline{\mathtt{r}}) \xrightarrow{rd\ \mathtt{r}} \mathcal{E}'\ [\overline{\mathtt{r}}/\overline{\mathtt{x}}, \mathtt{r}/\mathtt{this}]\mathtt{e}} \qquad \text{(R-Invk)}$$

$$\frac{\mathtt{r}\ \text{fresh} \quad \mathcal{E}' = \mathit{extend}(\mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{null}}), \mathcal{E})}{\mathcal{E}\ \mathtt{new\ C}() \xrightarrow{xt\ \mathtt{r}} \mathcal{E}'\ \mathtt{r}} \qquad \text{(R-New)}$$

Global Computation:

$$\frac{P = P'' \mid \mathtt{t}[\,\mathtt{e}\,] \qquad \mathcal{E}\ \mathtt{e} \xrightarrow{\alpha} \mathcal{E}'\ \mathtt{e}' \quad P' = P'' \mid \mathtt{t}[\,\mathtt{e}'\,] \quad \Gamma' = \mathit{reflect}(\mathtt{t}, \mathcal{E}', \Gamma)}{\Gamma\ P \xrightarrow{\alpha}_{\mathtt{t}} \Gamma'\ P'} \qquad \text{(G-Plain)}$$

$$\frac{P = P'' \mid \mathtt{t}[\,E[\,\mathtt{spawn\ e}\,]\,] \qquad P' = P'' \mid \mathtt{t}[\,E[\,\mathtt{null}\,]\,] \mid \mathtt{t}'[\,\mathtt{e}\,] \quad \mathtt{t}'\ \text{fresh} \quad \Gamma' = \mathit{spawn}(\mathtt{t}, \mathtt{t}', \Gamma)}{\Gamma\ P \xrightarrow{sp\ \mathtt{t}'}_{\mathtt{t}} \Gamma'\ P'} \qquad \text{(G-Spawn)}$$

$$\frac{P = P'' \mid \mathtt{t}[\,E[\,\mathtt{onacid}\,]\,] \qquad P' = P'' \mid \mathtt{t}[\,\mathtt{null}\,] \quad \mathtt{l}\ \text{fresh} \qquad \Gamma' = \mathit{start}(\mathtt{l}, \mathtt{t}, \Gamma)}{\Gamma\ P \xrightarrow{ac}_{\mathtt{t}} \Gamma'\ P'} \qquad \text{(G-Trans)}$$

$$\frac{\begin{array}{c}\Gamma = \Gamma'.\mathtt{t}, \mathcal{E} \quad \mathcal{E} = \mathcal{E}'.\mathtt{l} : \rho \\ \overline{\mathtt{t}} = \mathtt{t}_1 \ldots . \mathtt{t}_n = \mathit{intranse}(\mathtt{l}, \Gamma) \\ P = P'' \mid \overline{\mathtt{t}[\,E[\,\mathsf{commit}\,]\,]} \qquad P' = P'' \mid \overline{\mathtt{t}[\,E[\,\mathtt{null}\,]\,]} \\ \mathtt{t}_1, \mathcal{E}_1, \mathtt{t}_2, \mathcal{E}_2, \ldots, \mathtt{t}_n, \mathcal{E}_n \in \Gamma \qquad \overline{\mathcal{E}} = \mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n \\ \Gamma' = \mathit{commit}(\overline{\mathtt{t}}, \overline{\mathcal{E}}, \Gamma)\end{array}}{\Gamma\ P \xrightarrow{co}_{\mathtt{t}} \Gamma'\ P'} \qquad \text{(G-Comm)}$$

$$\frac{P = P' \mid \mathtt{t}[\,\mathtt{r}\,] \quad \Gamma = \mathtt{t}, \mathcal{E}\ .\ \Gamma'}{\Gamma\ P \xrightarrow{ki}_{\mathtt{t}} \Gamma'\ P'} \qquad \text{(G-ThKill)}$$

Fig. 5. TFJ Syntax and Semantics.

The metavariable $\mathtt{l}$ ranges over transaction names, and sequences of transaction names are used to represent the nesting structure. As usual, $\stackrel{\alpha}{\Longrightarrow}_{\mathtt{t}} *$ denotes the reflexive and transitive closure of the global reduction relation. The congruence rules given in Figure 6 are straightforward.

We work up to congruence of processes ($\mathtt{P}|\mathtt{P}' \equiv \mathtt{P}'|\mathtt{P}$, $(P_1|P_2)|P_3 \equiv \mathtt{P}_1|(P_2|P_3)$, and $\mathtt{P}|\mathtt{0} \equiv \mathtt{P}$). Congruence over expressions is defined in terms of evaluation contexts. An evaluation context is a term with a "hole" in it. The expression "picked" for evaluation (i.e., that fills the hole) is determined by the structure of these contexts. Observe that the main role of contexts is to enforce order of evaluation (e.g., left-to-write evaluation for method calls), and to ensure that subexpressions of $\mathtt{e}$ in $\mathtt{spawn}\ (\mathtt{e})$ are not chosen for evaluation prior to the evaluation of the $\mathtt{spawn}$ action to create a new thread. The other definitions are similar to those used in the specification of FJ. *fields* returns the list of all fields of a class including inherited ones. *mbody* returns the body of the method in a given class.

Let $\mathcal{E}$ be a transaction environment of the form $\mathtt{l}_0{:}\rho_0 \ldots \mathtt{l}_n{:}\rho_n$, then $\ell(\mathcal{E})$ extracts the order transaction label sequence, $\ell(\mathcal{E}) = \mathtt{l}_0 \ldots \mathtt{l}_n$ if $\Gamma = \mathtt{t}, \mathcal{E}\ .\ \Gamma'$. We override the definition of ℓ such that $\ell(\mathtt{t}, (\mathtt{t}, \mathcal{E}\ .\ \Gamma')) = \ell(\mathcal{E})$. The auxiliary function $\mathit{last}(\mathtt{r}, \rho)$ is defined to return a one element sequence containing the last value referenced by $\mathtt{r}$ in the environment ρ or the empty sequence if there is no binding for $\mathtt{r}$. It is defined inductively to return $\langle\,\rangle$ if $\rho = \langle\,\rangle$, $\mathtt{C}(\overline{\mathtt{r}})$ if $\rho = \rho'\ .\ \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}})$ and $\mathit{last}(\mathtt{r}, \rho')$ otherwise. The function $\mathit{first}(\mathtt{r}, \rho)$ is similar but returns the first binding for $\mathtt{r}$ in the sequence. Finally, $\mathit{findlast}(\mathtt{r}, \mathcal{E})$ finds the last binding for $\mathtt{r}$ in transaction environment $\mathcal{E}$.

There are four computational core reduction rules shown in in Fig. 5. (R-Field) evaluates a field access expression. (R-Assign) evaluates an assignment expression, (R-Invk) evaluates a method invocation expression and (R-New) evaluates an object instantiation expression. Notice that TFJ has a call-by-value semantics which requires that arguments be fully evaluated before performing method invocations or field access; the order in which arguments are evaluated in a call is unspecified. These rules are complemented by five global reduction rules. (G-Plain) corresponds to a step of computation, (G-

Evaluation Contexts:

$$E[\bullet] \mid E[\bullet].\mathtt{f} := \mathtt{e} \mid \mathtt{e}.\mathtt{f} := E[\bullet] \mid E[\bullet].\mathtt{m}(\overline{\mathtt{e}}) \mid \mathtt{e}.\mathtt{m}(\overline{\mathtt{r}}\ , E[\bullet]\ , \overline{\mathtt{e}})$$

Congruence:

$$\frac{\mathcal{E}\ \mathtt{e} \stackrel{\alpha}{\Longrightarrow} \mathcal{E}'\ \mathtt{e}'}{\mathcal{E}\ E[\mathtt{e}] \stackrel{\alpha}{\Longrightarrow} \mathcal{E}'\ E[\mathtt{e}']}$$

Transaction membership:

$$\frac{}{\mathit{nested}(\mathtt{l}, \langle\,\rangle) = \langle\,\rangle} \qquad \frac{\Gamma = \mathtt{t}, \mathcal{E}\ .\ \Gamma' \quad \mathit{nested}(\mathtt{l}, \Gamma') = \overline{\mathtt{t}} \quad \overline{\mathtt{l}}\ .\ \mathtt{l}\ .\ \mathtt{l}'\ .\ \overline{\mathtt{l}}' = \ell(\mathtt{t}, (\mathtt{t}, \mathcal{E}))}{\mathit{nested}(\mathtt{l}, \Gamma) = \mathtt{t}\overline{\mathtt{t}}} \qquad \frac{\Gamma = \mathtt{t}, \mathcal{E}\ .\ \Gamma' \quad \mathit{nested}(\mathtt{l}, \Gamma') = \overline{\mathtt{t}} \quad \mathtt{l} \notin \ell(\mathtt{t}, (\mathtt{t}, \mathcal{E})) \ \vee\ \overline{\mathtt{l}}\mathtt{l} = \ell(\mathtt{t}, (\mathtt{t}, \mathcal{E}))}{\mathit{nested}(\mathtt{l}, \Gamma) = \overline{\mathtt{t}}}$$

$$\frac{}{\mathit{intranse}(\mathtt{l}, \langle\,\rangle) = \langle\,\rangle} \qquad \frac{\Gamma = \mathtt{t}, \mathcal{E}\ .\ \Gamma' \quad \mathit{intranse}(\mathtt{l}, \Gamma') = \overline{\mathtt{t}} \quad \mathtt{l} \in \ell(\mathtt{t}, (\mathtt{t}, \mathcal{E})) \quad \mathit{nested}(\mathtt{l}, \Gamma) = \langle\,\rangle}{\mathit{intranse}(\mathtt{l}, \Gamma) = \mathtt{t}\overline{\mathtt{t}}} \qquad \frac{\Gamma = \mathtt{t}, \mathcal{E}\ \Gamma' \quad \mathit{intranse}(\mathtt{l}, \Gamma') = \overline{\mathtt{t}} \quad \mathtt{l} \notin \ell(\mathtt{t}, (\mathtt{t}, \mathcal{E}))}{\mathit{intranse}(\mathtt{l}, \Gamma) = \overline{\mathtt{t}}}$$

Fig. 6. Auxiliary definitions

Spawn) corresponds to a thread creation, (G-Trans) corresponds to the start of a new transaction, (G-Comm) corresponds to the commit of a transaction, and (G-ThKill) is a reclamation rule for threads in normal form. Most of the rules are straightforward. G-Plain makes use of a *reflect* operation that must propagate the action performed to other threads executing within this transaction. Notice that (G-Comm) requires that, if some thread `t` running in transaction `l` is ready to commit, all other threads executing in that transaction be ready to commit. The auxiliary predicate $intranse(\texttt{l}, \Gamma)$ given in Fig. 6 returns the set of threads that currently have the transaction label `l`. Note that if there is any thread running in a nested transaction (*e.g.*, has label $\texttt{l}\texttt{l}'$, for some `l`'), $intranse(\texttt{l}, \Gamma)$ will return the empty sequence as nested transactions must commit before their parent transaction. The (G-ThKill) rule takes care of removing threads that have terminated, to prevent blocking a transaction (terminated threads are not ready to commit).

The dynamic semantics leaves open the specification of a number of operations. In particular, the definitions of *read*, *write*, *spawn*, *extend*, *reflect*, *start*, and *join* are left unspecified. A particular incarnation of a transactional semantics must provide a specification for these operations.

Example To illustrate the rules, consider the term:

```
r.m(onacid, new C().m().f, commit)
```

To be faithful to the calculus syntax, sequencing of operations is expressed by the order of arguments in a method call. This expression creates an instance of class `C`, evaluates method `m` to yield an object that contains field `f`, and returns the reference denoted by that field. These actions are encapsulated within a transaction. The result of the transaction (here, the reference denoted by field `f`) is then supplied as an argument to method `m` accessed via reference `r`

We sketch the application of the semantic rules for this term. Assume that this term is evaluated by thread `t` in program state Γ, and process state P. By the congruence rule and definition of evaluation contexts, we can pick E to be

```
r.m(•, new C().m().f, commit)
```

and apply rule (G-Trans). Application of this rule yields a new term,

```
r.m(null, new C().m().f, commit)
```

and new program state Γ' defined to be $start(\texttt{l}, \texttt{t}, \Gamma)$; application of *start* results in the installation of a new transaction label `l` for thread `t` in its transaction environment. We can next "pick" • to be `new C()` (by the definition of the context grammar) to yield a new context term:

```
r.m(null, •.m().f, commit)
```

By application of rule G-Plain and rule R-New, the binding environment associated with `l` in `t`'s transaction environment is extended with a binding for fresh reference `v` to $\texttt{C}(\overline{\texttt{null}})$. The resulting term is:

```
r.m(null, v.m().f, commit)
```

Again, by the structure of evaluation contexts, we can pick E to be:

```
r.m(null, v.•.f, commit)
```

and apply rule G-Plain and R-Invk on term `m()`. Evaluating the body of `m` (not shown) will eventually yield a reference `v'` to an object containing field `f`. Thus, the context becomes

```
r.m(null, v.v'.•, commit)
```

and is further evaluated through the application of rules G-Plain and R-Field. The value yielded therein (call it `u`) results in the term:

```
r.m(null, u, commit)
```

We now apply rule G-Commit to yield a new state that reflects the commitment of this transaction, and a new term:

```
r.m(null, u null)
```

This expression can be now reduced by application of rule R-Invk.

5 Versioning Semantics

In Fig. 7 we define an instantiation of TFJ in which transactions implement sequences of object versions. The versioning semantics extends the notion of transaction environments to be an ordered sequence of pairs, each pair consisting of a transaction label and an environment. The intuition is that every transaction operates using a private *log*; these logs are treated as sequences of pairs, bindings a reference to its value. A log thus records effects that occur while executing within the transaction. A given reference may have different binding values in different logs. If $\mathcal{E} = \mathtt{l}_1{:}\rho_1 \ . \ \mathtt{l}_2{:}\rho_2$ then a thread `t` executing with respect to this sequence of transaction environments is evaluating expressions whose effects are recorded in log ρ_2 and which are part of the dynamic context of an `onacid` command with label $\mathtt{l}_2$. If $\mathtt{l}_2$ successfully commits, bindings in ρ_2 are merged with those in ρ_1. Once $\mathtt{l}_2$ commits, subsequent expressions evaluated by `t` occur within the dynamic context of an `onacid` command with label $\mathtt{l}_1$; effects performed by these expressions are recorded in environment ρ_1.

Thus, a transaction environment in a versioning semantics defines a chain of nested transactions: every $\mathtt{l}{:}\rho$ element in $\mathcal{E}$ is related to its predecessor in the sequence defined by $\mathcal{E}$ under an obvious static nesting relationship. A locus of computation can be therefore uniquely denoted by a thread `t` and the transaction label sequence $\overline{\mathtt{l}}$ in which `t` is executing.

When a new thread is created (cf. *spawn*), the global state is augmented to include the new thread; evaluation of this thread occurs in a transaction environment inherited from its parent. In other words, a spawned thread begins evaluation in the environment of its parent extant at the point where the thread was created.

When a thread enters a new transaction (cf. *start*), a new transaction environment is added to its state. This environment is represented as a pair consisting of a label denoting the transaction, and a log used to hold bindings for objects manipulated within the transaction. Initially, the newly created transaction is bound to an empty log.

The essence of the versioning semantics is captured by the *read*, *write*, and *commit* operations. If a *read* operation on reference `r` occurs within transaction `l`, the last value for `r` in the log is returned via the auxiliary procedure *findlast*, and the log associated with `l`

is augmented to include this binding. Thus, the first *read* operation for reference $\mathtt{r}$ within

$$\frac{\mathcal{E} = \mathcal{E}' \,.\, \mathtt{l}{:}\rho \quad \mathit{findlast}(\mathtt{r}, \mathcal{E}) = \mathtt{C}(\overline{\mathtt{r}}) \quad \mathcal{E}'' = \mathcal{E}' \,.\, \mathtt{l}{:}(\rho \,.\, \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}))}{\mathit{read}(\mathtt{r}, \mathcal{E}) \;=\; \mathcal{E}'', \mathtt{C}(\overline{\mathtt{r}})} \qquad \frac{\mathcal{E} = \mathcal{E}' \,.\, \mathtt{l}{:}\rho \quad \mathit{findlast}(\mathtt{r}, \mathcal{E}) = \mathtt{D}(\overline{\mathtt{r}}) \quad \mathcal{E}'' = \mathcal{E}' \,.\, \mathtt{l}{:}(\rho \,.\, \mathtt{r} \mapsto \mathtt{D}(\overline{\mathtt{r}}) \,.\, \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}))}{\mathit{write}(\mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}), \mathcal{E}) = \mathcal{E}''}$$

$$\frac{\mathcal{E} = \mathcal{E}' \,.\, \mathtt{l}{:}\rho \quad \mathcal{E}'' = \mathcal{E}' \,.\, \mathtt{l}{:}(\rho \,.\, \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}))}{\mathit{extend}(\mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}), \mathcal{E}) = \mathcal{E}''}$$

$$\frac{\Gamma = \mathtt{t}, \mathcal{E} \,.\, \Gamma' \quad \Gamma'' = \mathtt{t}', \mathcal{E} \,.\, \Gamma}{\mathit{spawn}(\mathtt{t}, \mathtt{t}', \Gamma) = \Gamma''} \qquad \frac{\Gamma = \mathtt{t}, \mathcal{E} \,.\, \Gamma' \quad \Gamma'' = \mathtt{t}, (\mathcal{E} \,.\, \mathtt{l}{:}\langle\,\rangle) \,.\, \Gamma}{\mathit{start}(\mathtt{l}, \mathtt{t}, \Gamma) = \Gamma''}$$

$$\frac{}{\mathit{reflect}(\mathtt{t}, \mathcal{E}, \langle\,\rangle) = \langle\,\rangle} \qquad \frac{\Gamma = \mathtt{t}, \mathcal{E}' \,.\, \Gamma' \quad \mathit{reflect}(\mathtt{t}, \mathcal{E}, \Gamma') = \Gamma''}{\mathit{reflect}(\mathtt{t}, \mathcal{E}, \Gamma) = \mathtt{t}, \mathcal{E} \,.\, \Gamma''} \qquad \frac{\Gamma = \mathtt{t}', \mathcal{E}' \,.\, \Gamma' \quad \mathit{reflect}(\mathtt{t}, \mathcal{E}, \Gamma') = \Gamma'' \quad \mathit{copy}(\mathcal{E}, \mathcal{E}') = \mathcal{E}'' \quad \Gamma''' = \mathtt{t}', \mathcal{E}'' \,.\, \Gamma''}{\mathit{reflect}(\mathtt{t}, \mathcal{E}, \Gamma) = \Gamma'''}$$

$$\frac{}{\mathit{commit}(\langle\rangle, \langle\rangle, \Gamma) = \Gamma} \qquad \frac{\begin{array}{c}\mathcal{E} = \mathcal{E}' \,.\, \mathtt{l}{:}\rho \quad \mathit{readset}(\rho, \langle\,\rangle) = \rho' \quad \mathit{writeset}(\rho, \langle\,\rangle) = \rho'' \\ \mathit{check}(\rho', \mathcal{E}') \quad \mathcal{E}' = \mathcal{E}'' \,.\, \mathtt{l}'{:}\rho''' \quad \mathit{reflect}(\mathtt{t}, (\mathcal{E}'' \,.\, \mathtt{l}'{:}\rho'''.\rho''), \Gamma) = \Gamma' \\ \mathit{commit}(\overline{\mathtt{t}}, \overline{\mathcal{E}}, \Gamma') = \Gamma''\end{array}}{\mathit{commit}(\mathtt{t}\overline{\mathtt{t}}, \mathcal{E}\overline{\mathcal{E}}, \Gamma) = \Gamma''}$$

$$\frac{\mathcal{E} = \mathtt{l}{:}\rho \,.\, \mathcal{E}'' \quad \mathcal{E}' = \mathtt{l}{:}\rho'' \,.\, \mathcal{E}'''}{\mathit{copy}(\mathcal{E}, \mathcal{E}') = \mathtt{l}{:}\rho \,.\, \mathit{copy}(\mathcal{E}'', \mathcal{E}''')} \qquad \frac{\mathcal{E} = \mathtt{l}'{:}\rho' \,.\, \mathcal{E}'' \quad \mathcal{E}' = \mathtt{l}{:}\rho \,.\, \mathcal{E}'''}{\mathit{copy}(\mathcal{E}, \mathcal{E}') = \mathtt{l}{:}\rho \,.\, \mathit{copy}(\mathcal{E}'', \mathcal{E}''')}$$

$$\frac{}{\mathit{check}(\langle\,\rangle, \mathcal{E})} \qquad \frac{\mathit{findlast}(\mathtt{r}, \mathcal{E}) = \mathtt{C}(\overline{\mathtt{r}}) \quad \mathit{check}(\rho, \mathcal{E})}{\mathit{check}(\rho \,.\, \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}), \mathcal{E})}$$

Mod sets:

$$\frac{}{\mathit{readset}(\langle\,\rangle, \_) = \langle\,\rangle} \qquad \frac{\rho = \mathtt{u} \mapsto \mathtt{C}(\overline{\mathtt{u}}) \,.\, \rho'' \quad \mathtt{u} \notin \overline{\mathtt{r}} \quad \mathit{readset}(\rho'', \overline{\mathtt{r}}\mathtt{u}) = \rho'}{\mathit{readset}(\rho, \overline{\mathtt{r}}) = \mathtt{u} \mapsto \mathtt{C}(\overline{\mathtt{u}'}) \,.\, \rho'}$$

$$\frac{\rho = \mathtt{u} \mapsto \mathtt{C}(\overline{\mathtt{u}}) \,.\, \rho'' \quad \mathtt{u} \in \overline{\mathtt{r}} \quad \mathit{readset}(\rho'', \overline{\mathtt{r}}) = \rho'}{\mathit{readset}(\rho, \overline{\mathtt{r}}) = \rho'}$$

$$\frac{}{\mathit{writeset}(\langle\,\rangle, \_) = \langle\,\rangle} \qquad \frac{\rho = \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}) \,.\, \rho'' \quad \mathit{writeset}(\rho'', \rho') = \rho''' \quad \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}) \neq \mathit{first}(\mathtt{r}, \rho')}{\mathit{writeset}(\rho, \rho') = \mathtt{u} \mapsto \mathtt{D}(\overline{\mathtt{u}}) \,.\, \rho'''}$$

Fig. 7. Versioning semantics

transaction `l` will bind a value for `r` computed by examining the logs of `l`'s enclosing transactions, choosing the binding value found in the one closest to `l`. Subsequent reads of `r` made within `l` will find a binding value within `l`'s log. Thus, this semantics ensures an isolation property on reads: once an object is read within transaction `l`, effects on that object performed within other transactions are not visible until `l` attempts to commit its changes.

The *write* operation is defined similarly. Note that *write* augments the log of the current transaction with two bindings, one binding the reference to its value prior to the assignment, and the other reflecting the effect of the assignment. The former binding is needed to guarantee transactional consistency. Consider a write to a reference `r` in transaction `l` which has not yet been read or written in `l`. The effects of this write can be made visible when `l` attempts to commit only if no other transaction has committed modifications to `r` in the interim between the time where the write occurred, and `l` attempts to commit. If this invariant were violated, the desired serialization semantics on transaction would fail to hold. The *extend* operation inserts a new binding in the current transaction's log; since the reference being bound is fresh, there is no existing binding in the parent transaction against which a consistency check must be made upon commit.

The *commit* operation is responsible for committing a transaction. In our versioning semantics, a commit results in bindings for objects written within a transaction's log to be propagated to its parent. In order for a commit of transaction `l` to succeed, it must be the case that the binding value of every reference read or written in `l` must be the same as its current value in `l`'s parent transaction. Satisfaction of this condition implies the absence of a data race between `l` and its parent or siblings. The *reflect* operation defined in *commit* makes visible the effects of `l` in all threads executing in `l`'s parent transaction; when used in a transaction-local action, it propagates the effects of the action to other threads executing within this same transaction.

The versioning semantics defined here is akin to an optimistic concurrency protocol in which the validity of reads and writes of references performed within a transaction `l` is determined by the absence of modifications to those references in transactions which commit between the time the first read or write of the reference takes place in `l` and the time `l` commits. For example, consider transaction $\texttt{l}_1$ that commits $\texttt{r}_1$, transaction $\texttt{l}_2$ that commits $\texttt{r}_2$ and transaction `l` that accesses both $\texttt{r}_1$ and $\texttt{r}_2$; a valid serialization of these transactions would commit $\texttt{l}_1$ prior to the first access of $\texttt{r}_1$ in $\texttt{l}_2$, and would commit $\texttt{l}_2$ prior to the first access of $\texttt{r}_2$ in `l`. Provided $\texttt{l}_2$ does not modify $\texttt{r}_1$, no atomicity or consistency invariants on these transactions would be violated.

6 Strict Two-phase locking

With slight alteration, the versioning semantics can be modified to support a two-phase locking protocol. Our semantics is faithful to a locking protocol in which locks are first acquired on objects on behalf of a transaction before the objects can be accessed, and released only when commit actions on transactions occur. In our semantics, a reader executing within transaction `l` can read an object provided that the lock on that object is held by transaction `l` or a prefix of `l`; every object is initially locked by the transaction in which it was created. A writer executing within transaction `l` can acquire exclusive

access to an object provided the object is currently locked by $\mathtt{l}$ or prefix of $\mathtt{l}$. To support locking, we define a unique transaction label $\mathtt{l_L}$ bound to a lock environment ρ_L; $\rho_L(\mathtt{r})$ maps $\mathtt{r}$ to the transaction label sequence which identifies the transaction that currently has exclusive access to $\mathtt{r}$. If $\overline{\mathtt{l}} = \mathtt{l_1}.\mathtt{l_2}\ldots\mathtt{l_n}$ is such a sequence, then any thread $\mathtt{t}$ executing within $\mathtt{l_p}$ where $n \leq p$ can acquire a lock for $\mathtt{r}$. Lock ownership is changed either because (a) the transaction in which a read or write action occurs is a prefix of the transaction which currently owns the lock, or (b) the lock is currently owned by a child transaction which is about to commit, and lock ownership must be transferred to the parent transaction. Unlike the versioning semantics presented earlier, commit actions always succeed since the manner in which locks are acquired ensure that no serializability violations ensue. As a consequence, there is no explicit notion of abort in this definition. Once a lock is acquired, the transaction has exclusive ownership until it commits, or a child attempts to access the object. Transactions implicitly abort if it reaches a stuck state; in this case, a deadlock would be modeled by a global state in which every thread is stuck, executing within a transaction that requires a lock held by another. The modifications necessary to support two-phase locking are shown in Fig. 8.

7 Soundness

Proving the soundness of a particular transactional facility requires relating it to desired serialization characteristics that dictate a transaction's ACID properties. For any abort-free program trace there must be a corresponding trace in which the transactions

$$\frac{\mathcal{E} = \mathtt{l}_L : \rho_L \,.\, \mathcal{E}' \quad \mathit{last}(\mathtt{r}, \rho_L) = \overline{\mathtt{l}}' \quad \overline{\mathtt{l}}' \lhd \overline{\ell}(\mathcal{E})}{\mathit{checklock}(\mathtt{r}, \mathcal{E}) = \mathtt{true}} \qquad \frac{\mathcal{E} = \mathtt{l}_L : \rho_L \,.\, \mathcal{E}' \quad \mathit{last}(\mathtt{r}, \rho_L) = \overline{\mathtt{l}}' \quad \overline{\mathtt{l}}' \lhd \overline{\ell}(\mathcal{E}) \quad \mathcal{E}'' = \mathtt{l}_L : (\rho_L.\mathtt{r} \mapsto \ell(\mathcal{E})) \,.\, \mathcal{E}'}{\mathit{acquirelock}(\mathtt{r}, \mathcal{E}) = \mathcal{E}''}$$

$$\frac{\mathcal{E} = \mathcal{E}' \,.\, \mathtt{l}{:}\rho \quad \mathit{findlast}(\mathtt{r}, \mathcal{E}) = \mathtt{C}(\overline{\mathtt{r}}) \quad \mathcal{E}'' = \mathcal{E}' \,.\, \mathtt{l}{:}(\rho \,.\, \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}})) \quad \mathit{checklock}(\mathtt{r}, \mathcal{E}) = \mathtt{true}}{\mathit{read}(\mathtt{r}, \mathcal{E}) \;=\; \mathcal{E}'', \mathtt{C}(\overline{\mathtt{r}})}$$

$$\frac{\mathit{findlast}(\mathtt{r}, \mathcal{E}) = \mathtt{D}(\overline{\mathtt{u}}) \quad \mathcal{E}' = \mathit{acquirelock}(\mathtt{r}, \mathcal{E}) \quad \mathcal{E}'' = \mathcal{E}' \,.\, \mathtt{l}{:}\rho \quad \mathcal{E}''' = \mathcal{E}'' \,.\, \mathtt{l}{:}(\rho \,.\, \mathtt{r} \mapsto \mathtt{D}(\overline{\mathtt{u}}) \,.\, \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}))}{\mathit{write}(\mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}), \mathcal{E}) = \mathcal{E}'''}$$

$$\frac{}{\mathit{commit}(\langle\rangle, \langle\rangle, \Gamma) = \Gamma} \qquad \frac{\mathcal{E} = \mathtt{l}_L{:}\rho_L \,.\, \mathcal{E}' \quad \rho'_L = \mathit{release}(\ell(\mathcal{E}), \rho_L) \quad \mathcal{E}'' = \mathtt{l}_L{:}\rho'_L \,.\, \mathcal{E}' \quad \mathit{reflect}(\mathtt{t}, \mathcal{E}, \Gamma) = \Gamma' \quad \mathit{commit}(\overline{\mathtt{t}}, \overline{\mathcal{E}}, \Gamma') = \Gamma''}{\mathit{commit}(\mathtt{t}\overline{\mathtt{t}}, \mathcal{E}\overline{\mathcal{E}}, \Gamma) = \Gamma'}$$

$$\frac{\mathcal{E}' \,.\, \mathtt{l} : \rho = \mathit{acquirelock}(\mathtt{r}, \mathcal{E}) \quad \mathcal{E}'' = \mathcal{E}' \,.\, \mathtt{l}{:}(\rho \,.\, \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}))}{\mathit{extend}(\mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}), \mathcal{E}) = \mathcal{E}''''} \qquad \frac{\rho_L = \rho'_L{:}\mathtt{r} \mapsto \overline{\mathtt{l}\mathtt{l}} \quad \rho''_L = \mathit{release}(\overline{\mathtt{l}\mathtt{l}}, \rho'_L) \,.\, \mathtt{r} \mapsto \overline{\mathtt{l}}}{\mathit{release}(\overline{\mathtt{l}\mathtt{l}}, \rho_L) = \rho''_L} \qquad \frac{\rho_L = \rho'_L{:}\mathtt{r} \mapsto \overline{\mathtt{l}'} \quad \overline{\mathtt{l}\mathtt{l}} \neq \overline{\mathtt{l}'} \quad \rho''_L = \mathit{release}(\overline{\mathtt{l}\mathtt{l}}, \rho'_L) \,.\, \mathtt{r} \mapsto \overline{\mathtt{l}'}}{\mathit{release}(\overline{\mathtt{l}\mathtt{l}}, \rho_L) = \rho''_L}$$

Fig. 8. Lock-based commitment semantics

executed serially, *i.e.* all concurrent transactions execute atomically wrt one another. The key idea is that we should be able to *reorder* any abort-free sequence of reduction steps into a sequence that yields the same final state and in which reduction steps taken on behalf of different parallel transactions are not interleaved. We proceed to formalize this intuitive definition.

The height of a transaction environment $\mathcal{E} = \mathtt{l}_0{:}\rho_0 \ldots \mathtt{l}_n{:}\rho_n$, written $|\mathcal{E}|$, is n. For a state Γ, $max(\Gamma)$ returns a thread transaction environment $\mathtt{t}, \mathcal{E}$ such that $\mathcal{E}$ is the environment with the largest height $|\mathcal{E}|$ in Γ.

Intuitively, a *well-defined* state is a state in which there are no conflicts. Or in other words, a well-defined state is a state in which, if all threads were to attempt to commit, no abort would occur. One way to express the notion of a successful commit is by comparing the state of a transaction's parent at the point the child is started (call it S_i), and its state when the child commits (call it S_f). If the objects modified by the child have the same value in both S_i and S_f, the child can commit its changes since no atomicity or serialization invariants with respect to the parent have been violated. We formalize this intuition thus:

Definition 1 (Well-defined). *Let $\Gamma = (t, \mathcal{E})\ .\ \Gamma'$. We say that state Γ is well-defined if Γ' is also well-defined and for $\mathcal{E} = \mathtt{l}_1{:}\rho_1 \ldots \mathtt{l}_n{:}\rho_n$, we have* $\mathrm{first}(\mathtt{r},\rho_j) = \mathrm{last}(\mathtt{r},\rho_{j-1})$ *if $2 \leq j \leq n$, and $\mathtt{r} \in Dom(\rho_{j-1}) \cap Dom(\rho_j)$.*

To define soundness properties, we introduce the notion of control and data dependencies. A dependency defines a relation on actions which can be used to impose structure of transition sequences. In other words, a well-defined transition sequence will be one in which dependencies are not violated, and thus define safe serial orderings. To define dependencies, we first formalize a notion of an action: given a transition $P\,\Gamma \stackrel{\alpha}{\Longrightarrow}_{\mathtt{t}} P'\,\Gamma'$, we say that the corresponding *action*, written $\mathcal{A}$ is $(\alpha, \mathtt{t}, \ell(\mathtt{t}, \Gamma))$.

Definition 2 (Control Dependency). *Define a preorder $\overset{c}{\rightsquigarrow}$ on actions such that $\mathcal{A}_1 \overset{c}{\rightsquigarrow} \mathcal{A}_2$ (read $\mathcal{A}_1$ is control-dependent on $\mathcal{A}_2$) if the following holds:*

1. $\mathcal{A}_1 = (\alpha, \mathtt{t}, \overline{\mathtt{l}})$ *and* $\mathcal{A}_2 = (sp\,\mathtt{t}, \mathtt{t}', \overline{\mathtt{l}})$.
2. $\mathcal{A}_1 = (co, \mathtt{t}, \overline{\mathtt{l}})$ *and* $\mathcal{A}_2 = (\alpha, \mathtt{t}', \overline{\mathtt{l}}')$ *where* $\alpha \in \{rd, wr, xt\}$ *and* $\overline{\mathtt{l}}' \lhd \overline{\mathtt{l}}$.
3. $\mathcal{A}_1 = (\alpha, \mathtt{t}, \overline{\mathtt{l}})$ *and* $\mathcal{A}_2 = (ac, \mathtt{t}', \overline{\mathtt{l}}')$ *where* $\overline{\mathtt{l}}' \lhd \overline{\mathtt{l}}$.

Actions within a thread have an obvious dependency on the *spawn* operation that creates the thread. A commit action has a control dependency with *read* and *write* actions performed within the transaction to be commited or any of its parents; our definition of nested transactions requires that parent actions are visible to their children. In particular, *read*, *write* and *extend* actions performed by a parent transaction are influenced by commit actions performed by a child. Finally, actions performed within a transaction $\mathtt{l}$ depend upon the *onacid* operation that creates the transaction in which they execute.

Definition 3 (Data Dependency). *Define a preorder $\overset{d}{\rightsquigarrow}$ on actions such that $\mathcal{A}_1 \overset{d}{\rightsquigarrow} \mathcal{A}_2$ (read $\mathcal{A}_1$ is data-dependent on $\mathcal{A}_2$) if $\mathcal{A}_1$ is either $(rd\,\mathtt{r}, \mathtt{t}, \overline{\mathtt{l}})$, $(wr\,\mathtt{rr}', \mathtt{t}, \overline{\mathtt{l}})$ or $(wr\,\mathtt{r}'\mathtt{r}, \mathtt{t}, \overline{\mathtt{l}})$, and $\mathcal{A}_2$ is either $(wr\,\mathtt{rr}'', \mathtt{t}', \overline{\mathtt{l}}')$ or $(xt\,\mathtt{r}, \mathtt{t}', \overline{\mathtt{l}}')$, with $\overline{\mathtt{l}}' \lhd \overline{\mathtt{l}}$.*

A read or write action in some transaction that effects locations has a data dependency with a write or extend action executed in a parent transaction that effects these same locations since all such effects induced by the parent are visible to its descendents.

The key property for our soundness result is the permutation lemma which describes the conditions under which two reduction steps can be permuted. Let $\mathcal{A}$ and $\mathcal{A}'$ be a pair of actions which are not related under a control or data dependency. We write $\mathcal{A} \overset{\overline{d}}{\rightsquigarrow} \mathcal{A}'$ and $\mathcal{A} \overset{\overline{c}}{\rightsquigarrow} \mathcal{A}'$ to mean action $\mathcal{A}$ has, respectively, no c-dependence or d-dependence on $\mathcal{A}'$.

Definition 4 (Independence). *Action $\mathcal{A}$ is independent of $\mathcal{A}'$ if $\mathcal{A} \overset{\overline{c}}{\rightsquigarrow} \mathcal{A}'$ and $\mathcal{A} \overset{\overline{d}}{\rightsquigarrow} \mathcal{A}'$.*

The permutation lemma states that any pair of actions which have no dependencies with one another can be permuted in a reduction sequence. This result follows intuitively since the reordering of these actions are inconsequential because neither action observes the effects produced by the other. In other words, the definition of data and control dependence ensures that permuting actions which no have dependency relation can never introduce a conflict, or fail to prevent a commit operation as a result of the reordering provided the same commit action would have succeed in the original (unpermuted) sequence. The permutation lemma ensures that the state of the computation is the same irrespective of the order of the reduction steps (though the intermediate steps may differ). This lemma is specific to a particular transactional facility semantics (*e.g.*, optimistic, two-phase locking) and must be proved with respect to this semantics.

Lemma 1 (Permute). *Assume that Γ and Γ' are well-defined, and let R be the two-step sequence of reductions $P\,\Gamma \overset{\alpha}{\Longrightarrow}_{\mathtt{t}} P_0\,\Gamma_0 \overset{\alpha'}{\Longrightarrow}_{\mathtt{t}'} P'\,\Gamma'$. Let $\mathcal{A}$ be $\mathcal{A}$ is independent of $\mathcal{A}'$ then there exists a two-step sequence R' such that R' is $P\,\Gamma \overset{\alpha'}{\Longrightarrow}_{\mathtt{t}'} P_1\,\Gamma_1 \overset{\alpha}{\Longrightarrow}_{\mathtt{t}} P'\,\Gamma'$.*

Proof. (Versioning semantics). We prove the lemma by cases on $\alpha', \mathtt{t}'$. The two most interesting cases are when α' is either rd or wr. We assume that $\overline{\mathtt{l}} = \ell(\mathtt{t}, \Gamma)$ and $\overline{\mathtt{l}}' = \ell(\mathtt{t}', \Gamma_0)$.

Case $\alpha' = rd\,\mathtt{r}'$ and $\overline{\mathtt{l}} \lhd \overline{\mathtt{l}}'$.

By definition of data dependence α can not be a wr or xt on $\mathtt{r}'$. Since nested transactions must commit before the parent, α cannot be a co action on $\overline{\mathtt{l}}$. By definition of control dependence, α cannot be an ac action on $\overline{\mathtt{l}}'$ or a sp action on $\mathtt{t}$. Since Γ' is well-defined, α is not a ki of $\mathtt{t}'$. There are five remaining cases for $\alpha, \mathtt{t}$:

1. α is $rd\,\mathtt{r}$ or α is $rd\,\mathtt{r}'$: by the definition of *read*, a read operation only extends the bindings in the latest transaction environment of $\mathtt{t}$ with the value of $\mathtt{r}$ or $\mathtt{r}'$. Thus, reordering two read operations has no visible effect.
2. α is $wr\,\mathtt{r}'\mathtt{r}''$ or $xt\,\mathtt{r}'$: *write* and *extend* operations add new bindings to the latest transaction environment of $\mathtt{t}$, and in the case of *write*, a binding that records the current value of the object. As $(\alpha, \mathtt{t}, \Gamma)$ is independent of $(\alpha', \mathtt{t}', \Gamma')$, by definition of data dependence it must be that $\mathtt{r} \neq \mathtt{r}'$, these effects on the log are not visible to any read on $\mathtt{r}$.

3. α is *sp* or *ki*: A spawn action augments the global state; by the definition of independence, the spawned thread is unrelated to the thread in which α' executes. Similar reasoning applies for thread termination.

Case $\alpha' = rd\ \mathtt{r}'$ and $\overline{\mathtt{l}}' \lhd \overline{\mathtt{l}}$.

Since *write* and *extend* actions performed by a child transaction are not visible to its parent, a *commit* action is the only action that a child can perform whose effects would be visible to a *read* action undertaken by the parent. By definition of control dependence, however, α can not be a *co* if $(\alpha, \mathtt{t})$ is independent of $(\alpha', \mathtt{t}')$.

Case $\alpha' = wr\ \mathtt{r}'\mathtt{r}''$ and $\overline{\mathtt{l}} \lhd \overline{\mathtt{l}}'$.

By definition of data dependence, α is neither a *wr* or *xt* action on $\mathtt{r}'$ or $\mathtt{r}''$. By definition of control dependence, α is not a *sp* of $\mathtt{t}'$, an *ac* of $\overline{\mathtt{l}}'$, or a *co* of $\overline{\mathtt{l}}$. The remaining cases for α are:

1. α is $rd\ \mathtt{r}$: by definition of *write*, changes made within a child transaction $\overline{\mathtt{l}}$ are not visible to the parent $\overline{\mathtt{l}}'$. Thus, a write operation to location $\mathtt{r}'$ performed by a child transaction can be permuted with any read operation performed by the parent.
2. α is $wr\ \mathtt{r}\mathtt{r}'''$ or $xt\ \mathtt{r}$: as above, write actions have no dependence with write or extend actions to different locations, and can thus be permuted with those actions as well.
3. α is *ac*: An action that creates a new transaction has no effect on $\alpha', \mathtt{t}'$ provided the transaction being created is not $\overline{\mathtt{l}}'$.
4. α is *sp* or *ki*: Actions that spawn or kill threads have no effect on α' provided the thread they manipulate is not t'.

Case $\alpha' = wr\ \mathtt{r}\mathtt{r}'$ and $\overline{\mathtt{l}}' \lhd \overline{\mathtt{l}}$.

By definition of data dependency, α can not be $rd\ \mathtt{r}$, $wr\ \mathtt{r}\mathtt{r}''$ or $wr\ \mathtt{r}'\mathtt{r}$. By definition of control dependency, α can not be a *co* of $\overline{\mathtt{l}}'$. Since no other child action is visible to the parent α can be permuted with any other action.

Having considered symmetric cases for read and write actions, note that *ac* of $\overline{\mathtt{l}}$ is not permutable with any action on $\overline{\mathtt{l}}$. Similarly, $sp, \mathtt{t}$ is not permutable with any action α_t. Since the pair of actions defining R are independent, such permutations are not considered. All other actions do not induce observable effects outside the transaction in which they occur and are thus permutable with actions performed in another transaction. □

Proof. (Locking semantics). We prove the lemma by cases on $\alpha', \mathtt{t}'$. As before, the two most interesting cases are when α' is either *rd* or *wr*. We assume that $\overline{\mathtt{l}} = \ell(\mathtt{t}, \Gamma)$ and $\overline{\mathtt{l}}' = \ell(\mathtt{t}', \Gamma_0)$.

Case $\alpha' = rd\ \mathtt{r}'$ and $\overline{\mathtt{l}} \lhd \overline{\mathtt{l}}'$.

By definition of data dependence α can not be a *wr* or *xt* on $\mathtt{r}'$. Since nested transactions must commit before the parent, α cannot be a *co* action on $\overline{\mathtt{l}}$. By definition of control dependence, α cannot be an *ac* action on $\overline{\mathtt{l}}'$ or a *sp* action on $\mathtt{t}$. Since Γ' is well-defined, α is not a *ki* of $\mathtt{t}'$. There are five remaining cases for $\alpha, \mathtt{t}$:

1. $\underline{\alpha \text{ is } rd\,\mathtt{r} \text{ or } \alpha \text{ is } rd\,\mathtt{r}'}$: A read operation extends the bindings in the transaction environment of $\mathtt{t}$ with the value of $\mathtt{r}$ or $\mathtt{r}'$, and checks the lock environment of the object read. Now, since we assume $P\,\Gamma \overset{\alpha}{\Longrightarrow}_{\mathtt{t}} P_0\,\Gamma_0 \overset{\alpha'}{\Longrightarrow}_{\mathtt{t}'} P'\,\Gamma'$, it must be the case that $\mathtt{r}$ and $\mathtt{r}'$ are locked by transactions $\overline{\mathtt{l}}''$ and $\overline{\mathtt{l}}'''$ such that $\overline{\mathtt{l}}'' \lhd \overline{\mathtt{l}}$ and $\overline{\mathtt{l}}''' \lhd \overline{\mathtt{l}}$. Since reads do not affect the lock environment, reordering read actions is permissible assuming the validity of the original transitions.
2. $\underline{\alpha \text{ is } wr\,\mathtt{r}\mathtt{r}'' \text{ or } xt\,\mathtt{r}}$: A *write* and *extend* adds a binding to the lock environment ρ_L such that $\rho_L(\mathtt{r}) = \mathtt{l}$. Since $\mathtt{r} \neq \mathtt{r}'$, the change in the lock environment reflecting the lock on $\mathtt{r}$ has no effect on the *read* on $\mathtt{r}'$.
3. $\underline{\alpha \text{ is } sp \text{ or } ki}$: As with the versioning semantics, a spawn action augments the global state; by the definition of independence, the spawned thread is unrelated to the thread in which α' executes. Similar reasoning applies for thread termination.

Case $\alpha' = rd\,\mathtt{r}'$ and $\overline{\mathtt{l}}' \lhd \overline{\mathtt{l}}$.

Since $P\,\Gamma \overset{\alpha}{\Longrightarrow}_{\mathtt{t}} P_0\,\Gamma_0 \overset{\alpha'}{\Longrightarrow}_{\mathtt{t}'} P'\,\Gamma'$, it must be the case that $\mathtt{r} \neq \mathtt{r}'$ or $\mathtt{r}$ is locked by transaction $\overline{\mathtt{l}}'' \lhd \overline{\mathtt{l}}'$. In either case, the actions are permutable. Suppose $\alpha = wr\,\mathtt{r}'$. Then, $\rho_L(\mathtt{r}') = \overline{\mathtt{l}}$. But, this would mean that the *read* on $\mathtt{r}'$ in $\overline{\mathtt{l}'}$ would not succeed by the definition of *checklock*. A similar argument applies for *extend*. Thus, α cannot be a *write* or *read* to $\mathtt{r}'$. Operations on locations other than $\mathtt{r}'$ can be permuted since consistency checks performe by *read* only examine locks on its argument. The permutability of α' with respect to *commit* follows from the definition of control dependency.

Case $\alpha' = wr\,\mathtt{r}'\mathtt{r}''$ and $\overline{\mathtt{l}} \lhd \overline{\mathtt{l}}'$.

By definition of data dependence, α is neither a wr or xt action on $\mathtt{r}'$ or $\mathtt{r}''$. By definition of control dependence, α is not a sp of $\mathtt{t}'$, an ac of $\overline{\mathtt{l}}'$, or a co of $\overline{\mathtt{l}}$. The remaining cases for α are:

1. $\underline{\alpha \text{ is } rd\,\mathtt{r}}$: by definition of data dependence $\mathtt{r} \neq \mathtt{r}'$. By definition of *write*, any update is protected by a lock which is not accessed by *read* since $\mathtt{r} \neq \mathtt{r}'$.
2. $\underline{\alpha \text{ is } wr\,\mathtt{r}\mathtt{r}''' \text{ or } xt\,\mathtt{r}}$: By definition of data dependence, $\mathtt{r} \neq \mathtt{r}'$. As above, write actions have no dependence with write or extend actions to different locations which are protected by different locks, and can thus be permuted with those actions as well.
3. $\underline{\alpha \text{ is } ac}$: An action that creates a new transaction has no effect on $\alpha', \mathtt{t}'$ provided the transaction being created is not $\overline{\mathtt{l}}'$.
4. $\underline{\alpha \text{ is } sp \text{ or } ki}$: Actions that spawn or kill threads have no effect on α' provided the thread they manipulate is not t'.

Case $\alpha' = wr\,\mathtt{r}\mathtt{r}'$ and $\overline{\mathtt{l}}' \lhd \overline{\mathtt{l}}$.

By definition of data dependency, α can not be $rd\,\mathtt{r}$, $wr\,\mathtt{r}\mathtt{r}''$ or $wr\,\mathtt{r}'\mathtt{r}$. By definition of control dependency, α can not be a co of $\overline{\mathtt{l}}'$. All other actions on other locations manipulate distinct locks that do not affect permutability.

The remaining cases are identical to their treatment under a versioning semantics. □

Next, we define the notion of a program trace which abstracts program and states.

Definition 5 (Program Trace). *Let R be the sequence of reductions $P_0\,\Gamma_0 \overset{\alpha_0}{\Longrightarrow}_{t_0} \ldots P_n\,\Gamma_n \overset{\alpha_n}{\Longrightarrow}_{t_n} P_{n+1}\,\Gamma_{n+1}$. The trace of the reduction sequence R, written $tr(R)$, is $(\alpha_0, \mathtt{t}_0, \overline{\mathtt{l}}_0) \ldots (\alpha_n, \mathtt{t}_n, \overline{\mathtt{l}}_0)$ assuming that $\overline{\mathtt{l}}_i = \ell(\mathtt{t}_i, \Gamma_i)$ for $0 \leq i \leq n$.*

A program trace is *serial* if for all pairs of reduction steps with the same transaction label ($\overline{\mathtt{l}}$), all reductions occurring between the two steps are taken on behalf of that very transaction or nested transactions ($\overline{\mathtt{l}} \lhd \overline{\mathtt{l}}'$).

Definition 6 (Serial Trace). *A program trace, $tr(R) = (\alpha_0, \mathtt{t}_0, \overline{\mathtt{l}}_0) \ldots (\alpha_n, \mathtt{t}_n, \overline{\mathtt{l}}_n)$ is serial iff $\forall\, i, j, k$ such that $0 \leq i \leq j \leq k \leq n$ and $\overline{\mathtt{l}}_i = \overline{\mathtt{l}}_k$ we have $\overline{\mathtt{l}}_i \lhd \overline{\mathtt{l}}_j$.*

We now present a soundness theorem which states that any sequence of reductions that ends in a well-defined state can be reordered so that its program trace is serial.

Theorem 1 (Soundness). *Let R be a sequence of reductions $P_0\,\Gamma_0 \overset{\alpha_0}{\Longrightarrow}_{t_0} \ldots P_n\,\Gamma_n \overset{\alpha_n}{\Longrightarrow}_{t_n} P_{n+1}\,\Gamma_{n+1}$. If Γ_{n+1} is well-defined, then there exists a sequence R' such that R' is $P_0\,\Gamma_0 \overset{\alpha'_0}{\Longrightarrow}_{t'_0} \ldots P'_n\,\Gamma'_n \overset{\alpha'_n}{\Longrightarrow}_{t'_n} P_{n+1}\,\Gamma_{n+1}$ and $tr(R')$ is serial.*

Proof. By induction on traces. We must show that for a serial trace $R = P_0\,\Gamma_0 \overset{\alpha_0}{\Longrightarrow}_{\mathtt{t}_0} \ldots P_n\,\Gamma_n$, if we take a step and have $R' = R \overset{\alpha}{\Longrightarrow}_{\mathtt{t}} P\,\Gamma$, where Γ is well-defined and R' is not serial, that R' can be permuted to an equivalent serial trace. Let $tr(R) = (\alpha_0, \mathtt{t}_0, \overline{\mathtt{l}}_0) \ldots (\alpha_n, \mathtt{t}_n, \overline{\mathtt{l}}_n)$. Since Γ_{n+1} is well-defined, we must show that $(\alpha_n, \mathtt{t}, \overline{\mathtt{l}_\mathtt{n}})$ is independent of $(\alpha_{n-1}, \mathtt{t}_{n-1}, \overline{\mathtt{l}_{\mathtt{n}-1}})$ in order to generate a permutation of R' that defines an equivalent serial trace. Suppose that $\mathcal{A}_n = (\alpha_n, \mathtt{t}, \overline{\mathtt{l}_\mathtt{n}})$ is not independent of $\mathcal{A}_{n-1} = (\alpha_{n-1}, \mathtt{t}_{n-1}, \overline{\mathtt{l}_{\mathtt{n}-1}})$. Then, it must be the case that either $\mathcal{A}_n \overset{d}{\rightsquigarrow} \mathcal{A}_{n-1}$ or $\mathcal{A}_n \overset{c}{\rightsquigarrow} \mathcal{A}n - 1$. Since R' is not serial, $\overline{\mathtt{l}_{\mathtt{n}-1}} \not\lhd \overline{\mathtt{l}}_n$. But, this contradicts the definition of a control dependency. If $\mathcal{A}_n \overset{d}{\rightsquigarrow} \mathcal{A}_{n-1}$, $\Gamma_{n+1} = \mathtt{t}, \mathcal{E}, \Gamma'$ and $\mathcal{E} = \mathcal{E}'.\overline{\mathtt{l}_{\mathtt{n}-1}}{:}\rho_{n-1}{:}\overline{\mathtt{l}_\mathtt{n}}{:}\rho_n$, then ρ's bindings for some object $\mathtt{r}$ must be extended, and differ from its bindings in ρ_{n-1}. But, then Γ_{n+1} could not be well-defined. This means that $\mathcal{A}_n$ is independent of $\mathcal{A}_{n-1}$ and by Lemma 1, they can be permuted. This process can be simply repeated until a serial trace is achieved.

8 Observations

We now informally relate the TFJ semantics and soundness results to the four ACID properties of database theory.

Atomicity requires that updates made by a thread be installed in the shared environment in a single atomic operation. Equivalently, from the point of view of other threads, all effects of a committing transaction become visible at the instant the transaction commits. In the versioning semantics of Section 5, atomicity is guaranteed by the *reflect* operation used when committing. Upon commit, *reflect* propagates all updates performed by the committed transaction instantly to the environments of all threads. In the locking semantics, changes performed by a transaction become visible when it releases its locks to its parent.

Consistency implies that transactions executed in isolation preserve the consistency of the database as defined by the application semantics.

The serialization theorem ensures that transactions preserve the *isolation* property since effects made within one transaction are not visible to another until a commit point.

Finally, *durability* requires that changes made by a transaction be persistent. Since our model does not include volatile data, we can assume that this property holds.

9 Related Work

The association of transactions with programming control structures has provenance in systems such as Argus [19, 22], Camelot [12] Avalon/C++ [11] and Venari/ML [16], and has also been studied for variants of Java, notably by Garthwaite [14] and Daynes [8–10].

Our permutation lemma is related to Lipton's notion of reductions [18] where left and right movers are actions that can be permuted. More recently Qadeer *et. al.* [23] have applied the same idea to generate method summaries for multi-threaded Java.

There is a large body of work that explores the formal specification of various flavors of transactions [20, 7, 15]. However, these efforts do not explore the semantics of transactions when integrated into a high-level programming language. Most closely related to our goals is the work of Black *et. al.* [3] and Chothia and Duggan [6]. The former presents a theory of transactions that specify atomicity, isolation and durability properties in the form of an equivalence relation on processes. Like our work, they present a soundness result that captures the intuitive notion of serializable actions. Beyond significant technical differences in the specification of the semantics, our results differ most significantly from theirs insofar as we present a stratified semantics for a realistic kernel language intended to express different concurrency control models within the same framework. We believe our formulation will be thus thus more useful to implementations and analyzes.

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 [5, 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.* [16] 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.

10 Conclusions

This paper develops a semantic framework for specifying nested and multithreaded transactions. This semantic framework is a basis for exploring the design space of transactional

programming language semantics. We believe that transactions can improve the performance of programs running on parallel architectures and that improve reliability by eliminating data races. We have introduced the TFJ calculus, an imperative and concurrent object calculus with call-by-value semantics and support for nested and multi-threaded transactions. The TFJ semantics is parameterized by the definition of the transactional facility, and in particular of the concurrency control protocol. This is important as different protocols have different observable behaviors and different performance characteristics. So being able to discuss these protocols in the same semantic framework allows one to compare them and argue about tradeoffs. In this paper we show two instantiations of TFJ with, respectively, a versioning-based optimistic model, and a pessimistic two-phase locking protocol. We have proven a general soundness theorem that relates the semantics of TFJ to a serializability property. The soundness result is parametric as well as it relies on a permutation lemma that must be proven for each instantiation of TFJ. While much works remains to be done to integrate transactions into mainstream languages, we believe that this paper is a step in that direction.

References

1. Malcolm Atkinson and Mick Jordan. A review of the rationale and architectures of PJama: a durable, flexible, evolvable and scalable orthogonally persistent programming platform. Technical Report TR-2000-90, Sun Microsystems Laboratories, June 2000.
2. Nick Benton, Luca Cardelli, and Cédric Fournet. Modern Concurrency Abstractions for C#. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 415–440. Springer-Verlag, 2002.
3. 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.
4. R. Bruni, C. Laneve, and U. Montanari. Orchestrating Transactions in the Join Calculus. In *13^{th} International Conference on Concurrency Theory*, 2002.
5. N. Busi, R. Gorrieri, and G. Zavattaro. On the Serializability of Transactions in JavaSpaces. In *ConCoord 2001, International Workshop on Concurrency and Coordination*, 2001.
6. Tom Chothia and Dominic Duggan. Abstractions for Fault-Tolerant Computing. Technical Report 2003-3, Department of Computer Science, Stevens Institute of Technology, 2003.
7. Panos Chrysanthis and Krithi Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.
8. Laurent Daynès. Implementation of automated fine-granularity locking in a persistent programming language. *Software—Practice and Experience*, 30(4):325–361, April 2000.
9. Laurent Daynès and Grzegorz Czajkowski. High-performance, space-efficient, automated object locking. In *Proceedings of the International Conference on Data Engineering*, pages 163–172. IEEE Computer Society, 2001.
10. Laurent Daynès and Grzegorz Czajkowski. Lightweight flexible isolation for language-based extensible systems. In *Proceedings of the International Conference on Very Large Data Bases*, 2002.
11. D. D. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery in Avalon/C++. *IEEE Computer*, 21(12):57–69, December 1988.
12. Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.
13. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

14. Alex Garthwaite and Scott Nettles. Transactions for Java. In Malcolm P. Atkinson and Mick J. Jordan, editors, *Proceedings of the First International Workshop on Persistence and Java*, pages 6–14. Sun Microsystems Laboratories Technical Report 96-58, November 1996.
15. Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Data Management Systems. Morgan Kaufmann, 1993.
16. Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, 16(6):1719–1736, November 1994.
17. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
18. Richard J. Lipton. Reduction: a new method of proving properties of systems of processes. In *Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 78–86. ACM Press, 1975.
19. B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.
20. Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan-Kaufmann, 1994.
21. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In P. Wegner G. Agha and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 4, pages 107–150. The MIT Press, 1993.
22. J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Massachusetts, 1985.
23. Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing procedures in concurrent programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–255. ACM Press, 2004.
24. Jan Vitek, Suresh Jagannathan, Adam Welc, and Antony L. Hosking. A semantic framework for designer transactions. In *Proceedings of the European Symposium on Programming*, pages 249–263. Springer-Verlag, 2004.