Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2008-038 June 23, 2008

# Safe Open-Nested Transactions Through Ownership

Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

# Safe Open-Nested Transactions Through Ownership

Kunal Agrawal I-Ting Angelina Lee Jim Sukha
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA

## ABSTRACT

Researchers in transactional memory (TM) have proposed open nesting as a methodology for increasing the concurrency of a program. The idea is to ignore certain "low-level" memory operations of an open-nested transaction when detecting conflicts for its parent transaction, and instead perform abstract concurrency control for the "high-level" operation that nested transaction represents. To support this methodology, TM systems use an open-nested commit mechanism that commits all changes performed by an open-nested transaction directly to memory, thereby avoiding low-level conflicts. Unfortunately, because the TM runtime is unaware of the different levels of memory, an unconstrained use of open-nested commits can lead to anomalous program behavior.

In this paper, we describe a framework of *ownership-aware* transactional memory which incorporates the notion of modules into the TM system and requires that transactions and data be associated with specific *transactional modules* or Xmodules. We propose a new *ownership-aware commit mechanism*, a hybrid between an open-nested and closed-nested commit which commits a piece of data differently depending on whether the current Xmodule owns the data or not. Moreover, we give a set of precise constraints on interactions and sharing of data among the Xmodules based on familiar notions of abstraction. We prove that ownership-aware TM has has clean memory-level semantics and can guarantee *serializability by modules*, which is an adaptation of multilevel serializability from databases to TM. In addition, we describe how a programmer can specify Xmodules and ownership in a Java-like language. Our type system can enforce most of the constraints required by ownership-aware TM statically, and can enforce the remaining constraints dynamically. Finally, we prove that if transactions in the process of aborting obey restrictions on their memory footprint, the *OAT* model is free from *semantic deadlock*.

## 1. INTRODUCTION

In the past few years, transactional memory [4] has been an active field of research. Transactional memory (TM) is meant to simplify concurrency control in parallel programming by providing a transactional interface for accessing memory; the programmer simply encloses the critical region inside an `atomic` block, and the TM system ensures that that section of code executes atomically. A TM system enforces atomicity by tracking the memory locations that each transaction in the system accesses, finding transaction conflicts,

This research is supported in part by NSF Grants CNS-0615215 and CNS-0540248 and a grant from Intel corporation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $5.00.

and aborting and possibly retrying transactions that conflict. TM guarantees that transactions are ***serializable*** [10], that is, transactions affect global memory as if they were executed one at a time in some order, even if in reality, several executed concurrently.

When using TM, one of the issues that programmers must deal with is the semantics of ***nested transactions***. When a TM system has ***closed-nested*** transactions [6], if a transaction $X$ contains a closed-nested transaction $Y$, after $Y$ commits, for the purpose of detecting conflicts the TM runtime considers any memory locations accessed by $Y$ as conceptually also being accessed by $X$. TM with closed-nested transactions guarantees that transactions are serializable at the level of memory. Researchers have observed, however, that closed nesting might unnecessarily restrict concurrency in transactional programs because it does not allow two "high-level" transactions to ignore conflicts due to accesses to shared "low-level" memory inside nested transactions.

To increase concurrency in transactional programs, researchers have proposed the methodology of ***open-nested transactions***. The open-nesting methodology incorporates the ***open-nested commit mechanism*** [5,8]. Conceptually, when an open-nested transaction $Y$ (nested inside transaction $X$) commits, $Y$ makes its changes directly to memory instead of propagating the changes to its parent $X$. Thus, the TM runtime no longer detects conflicts with $X$ due to memory accessed by $Y$. In this methodology, the programmer considers $Y$'s internal memory operations to be at a "lower level" than $X$; therefore $X$ should not care about the memory accessed by $Y$ when checking for conflicts. Instead, $Y$ must acquire an *abstract lock* based on the high-level operation that $Y$ represents and propagate this lock to $X$, so that the TM system can perform concurrency control at an abstract level. Also, with open nesting, if $X$ aborts, it may need to execute *compensating actions* to undo the effect of its committed open-nested transactions $Y$. Moss in [7] illustrates use of open nesting with an application that uses a B-tree. In [9], Ni et. al describe a software TM system that supports the open-nesting methodology.

An unconstrained use of the open-nested commit mechanism can lead to anomalous program behavior [1] that can be tricky to reason about. Since programmers must understand the open-nested commit mechanism to program using open nesting, at first glance, it might seem that using the open-nesting methodology is complicated. Although researchers have demonstrated specific examples that safely use an open-nested commit mechanism, the literature on TM offers relatively little in the way of formal programming guidelines which one can follow to have *provable* guarantees of safety when using open-nested commits. Moreover, since these working examples require only two levels of nesting, it is not obvious how one can correctly use open-nested commits in a program with more than two levels of abstraction.

We believe that one reason for the apparent complexity of open nesting is that the mechanism and methodology make different assumptions about memory. Consider a transaction $Y$ open-nested inside transaction $X$. The open-nesting methodology requires that $X$ ignore the "lower-level" memory conflicts generated by $Y$, while the open-nested commit mechanism will ignore *all* the memory operations inside $Y$. Say $Y$ accesses two memory locations $\ell_1$ and $\ell_2$, and $X$ does not care about changes made to $\ell_1$, but does care about $\ell_2$. The TM system can not distinguish between these two accesses, and will commit both in an open-nested manner, leading to anomalous behavior. In fact, specific uses of open nesting that researchers describe [3, 9] work because they exhibit a clean separation of the data accessed by an outer transaction and its (nested) inner transaction. For instance, in the TCC examples [3], the open-nested transactions are operations on a data structure, and the data structure "owns" memory needed for its implementation that can not be accessed by a user's application.

### *Contributions*

In this paper, we bridge the gap between memory-level mechanisms for open nesting and the high-level view by explicitly integrating the notions of *transactional modules* (Xmodules) and *ownership* into the TM system. We believe such an ownership-aware TM system allows programmers safely use the methodology of open nesting because the runtime's behavior more closely reflects the programmer's intent, and because

the additional structure imposed by ownership allows a language and runtime to enforce properties needed to provide provable guarantees of "safety" to the programmer. More specifically, the contributions of this paper are as follows:

1. We extend the theoretical framework from [1] to model the TM system with the modules and ownership, and suggest a concrete set of guidelines for sharing of data and interactions between Xmodules.
2. We describe how the Xmodules and ownership can be specified in a Java-like language and propose a type system that enforces the above mentioned guidelines in the programs written using this language extension.
3. We formally describe the operational model, called the *OAT* model, which uses a new ***ownership-aware commit mechanism***, which is a compromise between open-nested commit and closed-nested commit. An ownership-aware commit of a transaction $T$ commits a memory location globally if that location belongs to the module of $T$; otherwise, the read or write to the location is propagated up to $T$'s parent transaction. Unlike an ordinary open-nested commit, the ownership-aware commit treats memory locations differently depending on the Xmodule that owns the location. Note that the ownership-aware commit is still a mechanism; programmers must still use it in combination with abstract locks and compensating actions to get the full methodology.
4. We prove that the if a program follows the guidelines for data sharing and interactions between Xmodules, then the *OAT* model guarantees serializability by modules, which is a generalization of "serializability-by-levels" used in database transactions. Ownership-aware commit is the same as open nested commit if no module ever accesses data belonging to other modules. Therefore, one corollary of our theorem is that open-nested transactions are serializable when modules do not share data. This observation explains why researchers [3, 9] have found it natural to use open-nested transactions in the absence of sharing, in spite of the apparent pitfalls in the open-nested transaction semantics.
5. We prove that under certain restrictive conditions, the computation can not enter a semantic deadlock.

***Outline***

The rest of the paper is organized as follows. In Section 2, we briefly review the transactional computation framework [1], and explain how we extend this framework to formally incorporate Xmodules and ownership. In Section 3, we describe an example application and describe language constructs for specifying Xmodules and ownership. In Section 4, we describe the *OAT* model, an operational model for the TM system. In Section 5, we give a formal definition of serializability by modules, and proof-sketch that the *OAT* model guarantees this definition. In Section 6, we give conditions under which the *OAT* model does not exhibit semantic deadlocks.

## 2. COMPUTATIONS WITH MODULES

In this section, we formally define the structure of transactional programs with Xmodules and incorporate the concepts of Xmodules and ownership into the transactional computation framework described in [1]. First, we briefly review the framework. We then add Xmodules to this framework, and describe a way to restrict data sharing between transactions of different Xmodules using a "module tree" structure.

***Transactional Computations***

In the framework from [1], the execution of a program is modeled using a "computation tree" $\mathcal{C}$ that summarizes both the information about the control structure of a program and the nesting structure of transactions, and an "observer function" $\Phi$ which characterizes the behavior of memory operations. A program execution is assumed to generate a *trace* $(\mathcal{C}, \Phi)$.

A computation tree $\mathcal{C}$ is defined as an ordered tree with two types of nodes: ***memory-operation nodes*** $\texttt{memOps}(\mathcal{C})$ as leaves and ***control nodes*** $\texttt{spNodes}(\mathcal{C})$ as internal nodes. A memory operation $v$ satisfies the

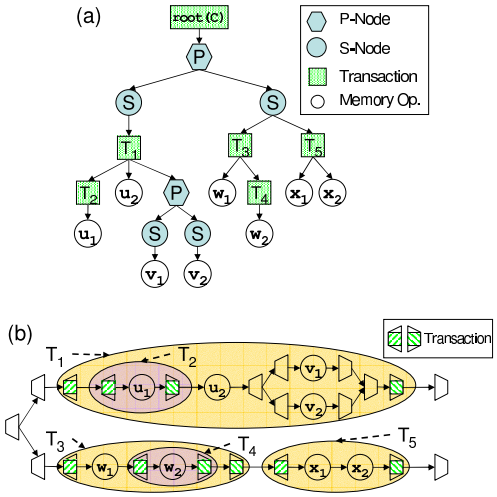


**Figure 1.** A sample (a) computation tree $\mathcal{C}$ and (b) its corresponding dag $G(\mathcal{C})$.

*read predicate* $R(v,\ell)$ if $v$ reads from location $\ell$, while $v$ satisfies the *write predicate* $W(v,\ell)$ if $v$ writes to $\ell$. Control nodes are either $S$ (series) or $P$ (parallel) nodes. Conceptually, the children of an $S$-node must be executed serially, from left to right, while the children of $P$ node can be executed in parallel. Some $S$ nodes are labeled as transactions; define $\texttt{xactions}(\mathcal{C})$ as the set of these nodes.

Instead of specifying the value that an operation reads or writes to a memory location $\ell$, we abstract away the values by using an ***observer function*** $\Phi$. For a memory operation $v$ that accesses a memory location $\ell$, the node $\Phi(v)$ is defined to be the operation that wrote the value of $\ell$ that $v$ sees.

We define several structural notations on the computation tree. Denote the ***root*** of a computation tree $\mathcal{C}$ as $\texttt{root}(\mathcal{C})$. For any tree node $X \in \texttt{nodes}(\mathcal{C})$, let $\texttt{ances}(X)$ denote the set of all ancestors of $X$ in $\mathcal{C}$, and let $\texttt{desc}(X)$ denote the set of all $X$'s descendants. Denote the set of proper ancestors of $X$ by $\texttt{pAnces}(X)$. Denote the ***least common ancestor*** of two nodes $X_1, X_2 \in C$ by $\texttt{LCA}(X_1, X_2)$. For any node $X \in \texttt{nodes}(\mathcal{C})$, we define the ***transactional parent*** of $X$, denoted $\texttt{xparent}(X)$, as $\texttt{parent}(X)$ if $\texttt{parent}(X) \in \texttt{xactions}(\mathcal{C})$, or $\texttt{xparent}(\texttt{parent}(X))$ if $\texttt{parent}(X) \notin \texttt{xactions}(\mathcal{C})$. Define the ***transactional ancestors*** of $X$ as $\texttt{xAnces}(X) = \texttt{ances}(X) \cap \texttt{xactions}(\mathcal{C})$. Define $\texttt{xLCA}(X_1, X_2)$ as $Z = \texttt{LCA}(X_1, X_2)$ if $Z \in \texttt{xactions}(\mathcal{C})$, and as $\texttt{xparent}(Z)$ otherwise.

A computation tree can also be represented as a computation dag (directed acyclic graph). Given a tree $\mathcal{C}$, the dag $G(\mathcal{C}) = (V(\mathcal{C}), E(\mathcal{C}))$ corresponding to the tree is constructed recursively. Every internal node $X$ in the tree appears as two vertices in the dag. Between these two vertices, the children of $X$ are connected in series if $X$ is an $S$ node, and are connected in parallel if $X$ is a $P$ node. Figure 1 show a computation tree and its corresponding computation dag.

In classical theories on transactions and serializability, a particular execution order for a program is referred to as a ***history*** [10]. In our framework, a history corresponds to a topological sort $\mathcal{S}$ of the computation dag $G(\mathcal{C})$. We define our models of TM using these sorts. Reordering a history to produce a serial history is equivalent to choosing different topological sorts $\mathcal{S}'$ of $G(\mathcal{C})$ whose observer function is still "consistent" with $\mathcal{S}'$, but where all transactions appear contiguous in $\mathcal{S}'$.

### *Xmodules and Computation Tree*

In this paper, we consider traces generated by a program which is organized into a set $\mathcal{N}$ of Xmodules. Each Xmodule $A \in \mathcal{N}$ has some number of methods and a set of memory locations associated with it. In the transactional computation framework, we assume every method of an Xmodule $A$ generates some transaction instance $T$. We use the notation $\texttt{xMod}(T) = A$ to associate the instance $T$ with the Xmodule $A$. We also define

the instances associated with $A$ as

$$\texttt{modXactions}(A) = \{T \in \texttt{xactions}(\mathcal{C}) \ : \ \texttt{xMod}(T) = A\}.$$

We partition the set of all memory locations $\mathcal{L}$ into sets of memory owned by each Xmodule. Let $\texttt{modMemory}(A) \subseteq \mathcal{L}$ denote the set of memory locations owned by $A$. For a location $\ell \in \texttt{modMemory}(A)$, we say that $\texttt{owner}(\ell) = A$. Xmodules of a program are arranged as a rooted, ordered tree called the ***module tree***, denoted by $\mathcal{D}$. The root of $\mathcal{D}$ is called the `world` module. An Xmodule $A$ is said to be owned by its parent $\texttt{modParent}(A)$ in $\mathcal{D}$. The set of ancestors of $A$ is $\texttt{modAnces}(A)$ ($\texttt{modDesc}(A)$ for descendants).

Each Xmodule is assigned an `level` according to its position in the tree as follows: visit the nodes in a left-to-right depth-first search order and assign ids in a descending order. Therefore `world` has the maximum `level`. Lower-level Xmodules have lower `level` numbers.

We use the module tree $\mathcal{D}$ to restrict the sharing of data between Xmodules and to limit the visibility of Xmodule methods according to the rules given in Definition 1.

**DEFINITION 1.** *A program with a module tree $\mathcal{D}$ should generate only traces $(C, \Phi)$ which satisfy the following rules:*

1. *For any memory operation $v$ which accesses a memory location $\ell$, let $T = \texttt{xparent}(v)$. Then $\texttt{owner}(\ell) \in \texttt{modAnces}(\texttt{xMod}(T))$.*
2. *Let $X, Y \in \texttt{xactions}(\mathcal{C})$ be transaction instances such that $\texttt{xMod}(X) = A$ and $\texttt{xMod}(Y) = B$. We can have $X = \texttt{xparent}(Y)$ only if $\texttt{modParent}(B) \in \texttt{modAnces}(A)$, and $\texttt{level}(A) > \texttt{level}(B)$.*

By Rule 1, an Xmodule $A$ can only directly access memory that it owns, or memory that an ancestor Xmodule $B$ owns (e.g., because $B$ passed in that data to a lower-level Xmodule). Since all ancestors of $A$ have higher `level` than $A$, a transaction from module $A$ can not directly access any "lower-level" memory.

Rule 2 says that a method from $A$ can call a method from $B$ only if $B$ is the child of some ancestor of $A$, and if $B$ is "to the right" of $A$ in the tree. The second rule requires that an Xmodule can only call methods of some (but not all) lower-level Xmodules.

In our model, primarily for convenience, we assume an method in an Xmodule $A$ never calls another transactional method from $A$ or an ancestor of $A$. If a method from $A$ does call another transactional method from $A$, the new method call does not generate a new transaction instance and we subsume the nested method call using flat nesting. Similarly, if a method from $A$ calls a method from an ancestor Xmodule (e.g., callback), we subsume the nested method call, and model this case as $A$ accessing the memory from ancestor Xmodule directly.[1]

The concept of higher-level and lower-level modules is inherent to the definition of serializability-by-modules and abstract serializability; the very justification of open-nesting is that transactions must be able to ignore lower-level conflicts. Therefore, our formalism requires a partial order among Xmodules; if an Xmodule $A$ can call Xmodule $B$, then conceptually $A$ is at a higher level than $B$. Therefore, $B$ can not call $A$ (except in a flat-nested manner described in the previous paragraph), since lower-level modules can not call methods from higher-level modules transactionally. If two components of the program call each other, then we would require that these two components be combined into the same Xmodule.

### *Properties of Xmodules*

Definition 1 guarantees certain properties of the computation tree which are essential to the ownership-aware commit mechanism. The following lemma can be proved by induction on nesting depth of transactions.

**LEMMA 1.** *Given a computation tree $\mathcal{C}$, consider any $T \in \texttt{xactions}(\mathcal{C})$. Let $S_T = \{\texttt{xMod}(T') \ : \ T' \in \texttt{xAnces}(T)\}$. Then $\texttt{modAnces}(\texttt{xMod}(T)) \subseteq S_T$.*

[1] One could also use closed nesting instead of flat nesting when an Xmodule calls its own methods or its ancestor's methods.

PROOF. We prove this fact by induction on the nesting depth of transactions in the computation tree.

In the base case, the top-level transaction $T = \texttt{root}(\mathcal{C})$, and $\texttt{xMod}(\texttt{root}(\mathcal{C})) = \texttt{world}$. Thus, the fact holds trivially.

For the inductive step, assume that $\texttt{modAnces}(\texttt{xMod}(T)) \subseteq S_T$ holds for any transaction $T$ at depth $d$. We show that the fact holds for any $T^* \in \texttt{xactions}(\mathcal{C})$ at depth $d+1$.

For any such $T^*$, we know $T = \texttt{xparent}(T^*)$ is at depth $d$. By Rule 2, $\texttt{modParent}(\texttt{xMod}(T^*)) \in \texttt{modAnces}(\texttt{xMod}(T))$. Thus, $\texttt{modAnces}(\texttt{xMod}(T^*)) \subseteq \texttt{modAnces}(\texttt{xMod}(T)) \cup \{\texttt{xMod}(T^*)\}$. By construction of the set $S_T$, we have $S_{T^*} = S_T \cup \{\texttt{xMod}(T^*)\}$. Therefore, we have $\texttt{modAnces}(\texttt{xMod}(T^*)) \subseteq S_{T^*}$. □

**THEOREM 2.** *If a transaction $T \in \texttt{xactions}(\mathcal{C})$ directly (without nesting) accesses a memory location $\ell$, then there exists a unique transaction $T^* \in (\texttt{xAnces}(T) - \{\texttt{root}(\mathcal{C})\}))$, such that*

1. $\texttt{owner}(\ell) = \texttt{xMod}(T^*)$, *and*
2. *For all transactions $X \in \texttt{pAnces}(T^*) \cap \texttt{xactions}(\mathcal{C})$, $X$ can not directly access location $\ell$.*

PROOF. This result follows from the properties of the module tree and computation tree stated in Definition 1.

First, by Rule 1, we know $\texttt{owner}(\ell) \in \texttt{modAnces}(\texttt{xMod}(T))$, i.e., $\ell$ is owned by some Xmodule which is an ancestor of $\texttt{xMod}(T)$ in the module tree. By Lemma 1, we know $\texttt{modAnces}(\texttt{xMod}(T)) \subseteq S_T$. Therefore, there exists some transaction $T^* \in \texttt{xAnces}(T)$ such that $\texttt{owner}(\ell) = \texttt{xMod}(T^*)$.

We can use Rule 2 to show that the $T^*$ is unique. Let $X_i$ be the chain of ancestor transactions of $T$. More formally, let $X_0 = T$, and let $X_i = \texttt{xparent}(X_{i-1})$, up until $X_k = \texttt{root}(\mathcal{C})$. By Rule 2, we know $\texttt{level}(\texttt{xMod}(X_i)) > \texttt{level}(\texttt{xMod}(X_{i-1}))$, that is, the module ids become strictly larger walking up the tree from $T$. Thus, there can only be one ancestor transaction $T^*$ of $T$ with $\texttt{level}(\texttt{xMod}(T^*)) = \texttt{level}(\texttt{owner}(\ell))$.

To check the second condition on $T^*$, consider any $X \in \texttt{pAnces}(T^*) \cap \texttt{xactions}(\mathcal{C})$, and assume for contradiction that $X$ could access $\ell$ directly. By Rule 1, $X$ can access $\ell$ directly only if $\texttt{owner}(\ell) \in \texttt{modAnces}(\texttt{xMod}(X))$, which then implies $\texttt{level}(\texttt{owner}(\ell)) \geq \texttt{level}(\texttt{xMod}(X))$, since an Xmodule always has a smaller id than its ancestor Xmodules. This, however, contradicts the facts derived earlier, that $\texttt{owner}(\ell) = T^*$ and $\texttt{level}(T^*) < \texttt{level}(\texttt{xMod}(X))$. □

Intuitively, Theorem 2 implies that for programs that obey the constraints described in Definition 1, if a transaction $T$ accesses a memory location $\ell$, then some unique ancestor of $T$, say $T^*$, belongs to the Xmodule that owns $\ell$. In the context of the ownership-aware commit mechanism, this transaction $T^*$ is "responsible for" committing $\ell$ and making it visible to the world. The second condition of Theorem 2 states that no ancestor transaction of $T^*$ in the call stack can ever directly access $\ell$; thus, it is "safe" for $T^*$ to commit $\ell$.

## 3. OWNERSHIP TYPES FOR Xmodules

In this section, we illustrate how one may use an ownership-aware transaction system to write a simple example application. First, we describe the example application, which consists of user code interacting with a simple database system. Next, we describe one way to split this application into Xmodules, and explain the restrictions imposed by Definition 1 in the context of this application. Finally, we describe language constructs for Java that can be used to both specify Xmodules and ownership for this application, and describe a type system design (called the *OAT* type system) that statically enforces some of the restrictions of Definition 1.

### *Example Application*

To explain the notions of modules and ownership, we describe an application similar to the one in [7], but extended to include more than two levels of transaction nesting and data sharing between a nested transaction and its parent.

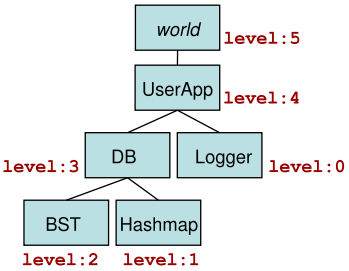


**Figure 2.** A module tree $\mathcal{D}$ for the program described in Section 1. The `level`'s are assigned by visiting Xmodules in a left-to-right depth-first tree walk, numbering Xmodules in a descending order.

Consider a user application which concurrently accesses a database of many individuals' book collections. The user application may provide many other functionalities in addition to accessing the book database, but for the purpose of this paper, we are only describing a subpart of a complex system.

The database implementor chooses to store records in a binary search tree, keyed by name. Each tree node corresponds to a different person, and maintains a list of books in that person's collection. The database supports queries by name, as well as updates that add a new person or a new book to a person's collection. The database also maintains a private hashmap, keyed by book title, to support a reverse query, i.e., given a book title, return a list of people who own the book.

Finally, the user application wants the database to log changes on disk for recoverability. Whenever the binary search tree or hash table are updated, the database inserts metadata into the buffer of a logger to record the change that just took place. Periodically, the user application is able to request a checkpoint operation which flushes the buffer to disk.

One may implement this example in Java with the following classes: `UserApp` as the top-level application that manages the book collections, `Person` and `Book` as the abstractions representing book owners and books, `DB` for the database, `BST` and `Hashmap` for the binary search tree and hashmap maintained by the database, and `Logger` for logging the metadata to disk. In addition, there are some other auxiliary classes such as tree node `BSTNode` for the `BST`, `Bucket` in the `Hashmap`, and `Buffer` used by the `Logger`.

### *Xmodules for Example Application*

Intuitively, an Xmodule is as a stand-alone entity that contains data and methods; a Xmodule owns data that it privately manages, and uses its methods to provide public services to other modules. Not all of a program's classes are meant to be Xmodules; some classes only wrap data, while others are Xmodules that provide services. In our example, we identify five Xmodules– `UserApp`, `DB`, `BST`, `Hashmap`, and `Logger`. The `UserApp` uses services from `DB`, `BST` and `Hashmap` are submodules of `DB`, and `Logger` provides services to all `UserApp`, `DB`, `Hashmap`, and `BST`. Classes such as `Book` and `Person`, on the other hand, are data types used by `UserApp`. Similarly, classes like `BSTNode` and `Bucket` are data types used by `BST` and `Hashmap` to maintain the internal state of the data structures.

We organize the Xmodules of the application into the module tree shown in Figure 2. `UserApp` is directly owned by `world`, `DB` and `Logger` are owned `UserApp`; `BST` and `Hashmap` are owned by `DB`. By dividing Xmodules this way, the ownership of data falls out naturally, i.e., an Xmodule owns certain pieces of data if the data is encapsulated under the Xmodule. For example, the instances of `Person` or `Book` are owned by `UserApp` because they should only be accessed either `UserApp` or its descendants.

If Definition 1, Rule 1 is satisfied, all of `DB`, `BST`, `Hashmap`, and `Logger` can only directly access data owned by `UserApp`, but the `UserApp` can not directly access data owned by any of the other Xmodules. This rule corresponds to standard software-engineering rules for abstraction; the "high-level" Xmodule `UserApp`

can pass its data down and lower-level Xmodules can access that data directly, but `UserApp` itself should not modify data owned by lower-level Xmodules.

If Rule 2 is satisfied, the `UserApp` may invoke methods from `DB`, `DB` may invoke methods from `BST` and `Hashmap`, and every other Xmodule may invoke methods from `Logger`. While the `BST` Xmodule can call methods from `Logger`, it can not pass data owned by itself directly into the `Logger`. But it can pass data owned by the `UserApp` to the logger, which is all that is required in this application. In the module tree in Figure 2, if the `Logger` had any children, then they would be lower level than `BST`, but `BST` can not call methods from this hypothetical child.

***Specification of Xmodules and Parametric Ownership Types***

Angelina: Ok, maybe the title

Although the restrictions on Xmodules required by Definition 1 are not difficult to state or reason about abstractly, the programmer has to specify the Xmodules and ownership of data in their programs. In addition, if the program violates the rules from Definition 1, then the compiler or the runtime system should be able report this error. We propose the *OAT* type system, which is an extension of the ownership type scheme of Boyapati et. al [2], because the restrictions described in Definition 1 are similar to the concept of object containment / encapsulation in an object-oriented language. Note that the scheme of Boyapati et. al allows owner polymorphism by parameterizing class / method declarations with ownership tags. We adapt this annotation as well to enable code reuse.

Before describing how to specify Xmodules and their corresponding data, we first describe the scheme of Boyapati et. al [2]. Their type system enforces the following properties:

1. Every object has a unique owner.
2. The owner can be either another object, or `world` .
3. The ownership relation forms an *ownership tree* rooted at `world`.
4. An object *a* can access another object *b* directly (*a* can obtain a pointer to *b*) only if *b* is either *a*'s child or *a*'s ancestor's child in the ownership tree.

They enforce these properties by adding annotations to class definitions and type declarations. Every type `T1` has a set of associated ownership tags, denoted $\texttt{T1}\langle f_1, f_2, \ldots f_n\rangle$. The first formal $f_1$ denotes the owner of the corresponding `this` object. The remaining formals $f_2, f_3, \ldots f_n$ are additional tags which the object can propagate down to its encapsulated objects. The formals get assigned with actual owners $o_1, o_2, \ldots o_n$ when an object *a* of type $T1$ is created. The type system checks that *a*'s owner $o_1$ is a descendant of $o_i$, $\forall i \in 2..n$, (denoted by $o_1 \preceq o_i$ henceforth) in the ownership tree. Of course, when an assignment takes place, the type system also enforces that the types from both sides match exactly.

Within the class definition of type `T1`, the only visible ownership tags are $\{f_1, f_2, \ldots f_n\} \cup \{\texttt{this}, \texttt{world}\}$, where `this` denotes the owner to be the corresponding `this` object, and `world` denotes the object to be globally accessible. The object can declare (and thereby access) another object of type `T2` using only owners from this set. Thus, an object can not access another object *b* if *b*'s owner is not *a* or one of *a*'s ancestors.

Boyapati et. al's type system enforces constraints on objects which are similar to, but not exactly the constraints that we would like for Xmodules (i.e., Definition 1). Therefore, we extend their type system to satisfy three additional requirements.

First, the *OAT* type system imposes restrictions to guarantee that only Xmodules own other objects. Normally, in the ownership tree of [2], every object can be an owner of other objects. Therefore, we explicitly distinguish between objects and Xmodules by requiring that Xmodules extend from a special `Xmodule` class. In addition, the *OAT* type system allows the use of `this` as an ownership tag only in the class definition that is a subtype of `Xmodule`.

Second, the *OAT* type system prohibits an Xmodule from having any primitive-type fields. In the parametric type system we use, one can not specify the owner of primitive fields of an object, and primitive fields are owned by the owner of the corresponding object. Thus, any primitive fields of an Xmodule *A* are

```
 1   public class UserApp<appO> extends Xmodule {
 2       private DB<this[0], this[1], this[2]> db;
 3       private Logger<this[1], this[2]> logger;
          ...
 4       public UserApp() {
 5           logger = new Logger<this[1], this[2]>();
 6           db = new DB<this[0], this[1], this[2]>(logger);
 7       }
 8   }

 9   public class DB<dbO, logO, dataO>
          extends Xmodule where(logO <= dataO) {
10       private Logger<logO, dataO> logger;
11       private BST<this[0], logO, dataO> bst;
12       private Hashmap<this[1], logO, dataO> hashmap;
13       public DB(Logger<logO, dataO> logger) {
14           this.logger = logger;
              ...
15   }
```

**Figure 3.** Specifying Xmodules and ownership for the example application described in Section 1.

owned by *A*'s parent in the ownership tree. Therefore, two sibling Xmodules would be able to access each other's primitive fields directly, since they have the same owner. To disallow this behavior, we do not allow Xmodules to declare primitive fields.

Lastly, the *OAT* type system enforces ordering between sibling Xmodules *A* and *B* to prevent cyclic dependencies between the subtrees of *A* and *B* in the module tree. In Boyapati's type system, an object can call any of its ancestor's siblings, while Definition 1 dictates that an Xmodule *A* can only call its ancestor's siblings to the right. To enforce this restriction, we extend each ownership tag *o* with an *index*, $o.index$.

Inside a class file for an Xmodule *A*, whenever the programmer wishes to specify an owner of `this`, the programmer must also specify a static index, i.e., pass in `this[i]` as the tag. The tag `this[i]` replaces some formal tag *o* in the type of $B_i$, and the index *i* becomes $o.index$, the index of the tag *o*. The type system uses these indices to impose a partial order on the children of *A* in the module tree. In this example, by specifying indices, the type system can statically enforce that $B_i$ never call a method from $B_j$ if $j < i$.

For the same reason, we disallow arbitrary use of the `world` tag; otherwise it would be difficult to enforce an ordering between sibling Xmodules owned directly by the `world`. Instead, we allow only the `main` method for the application program to specify owners using `world[i]` (with an index), thereby imposing an ordering among children belong to the `world`.

With these restrictions, the ownership tree in our system will only have Xmodules as internal nodes, and all other objects as leaves. Note that in our ownership tree, a parent-child relationship has two meanings. If an Xmodule *A* has a regular object *o* as its child, then *A* owns all the memory associated with *o*. When an Xmodule *A* has another Xmodule *B* as its child, *B* is *A*'s child in the Xmodule tree. The Xmodule tree does not contain objects.

Figure 3 illustrates how one can specify Xmodules and ownership using ownership types. The programmer specifies an Xmodule by creating a class which extends from a special `Xmodule` class. The `DB` class has three formal owner tags – `dbO` which is the owner of the `DB` Xmodule instance, `logO` which is the owner of the `Logger` Xmodule instance that the `DB` Xmodule will use, and one owner `dataO` for the user data being stored in the database. When an instance of `UserApp` initializes Xmodules in lines 5–6, it declares itself as the owner of the `Logger`, the `DB`, and the user data being passed into `DB`. The indices on `this` are declaring the ordering of Xmodules in the module tree, i.e., the user data is lower-level than the `Logger`, and the `Logger`

is lower level than the DB. lines 10–12 illustrate how the DB class can initialize its Xmodules and propagate the formal owner tags (i.e., log0 and data0) down.

***Type System Guarantees***

We extend the type system of [2] to encompass the requirements described in the previous section. To state the guarantees of our type system, we first define a partial order on indexed ownership tags.

**DEFINITION 2.** *For ownership tags with indices, we adopt the notation $o_1 \rhd o_2$ to mean that either $o_1 \preceq o_2$ and $o_1 \neq o_2$, or $o_1 = o_2$ and $o_1.index \leq o_2.index$.*

Note that if $A$ has owner tag $o_1$, $B$ has owner tag $o_2$, $o_1 = o_2$, and $o_1.index < o_2.index$, then $o_1$ and $o_2$ represent the same Xmodule instance, and $A$ and $B$ are sibling Xmodules, with $B$ to the right of $A$ in the module tree.

In summary, type system enforces the following properties.

1. The tag this[i] can be used as an ownership tag only in the class file of an Xmodule object.
2. Xmodule objects can not have primitive-type fields.
3. For a type $\texttt{T}\langle o_1, o_2, \ldots o_n\rangle$, we must have $o_1 \rhd o_i$ for all $i \in \{2, \ldots n\}$.
4. A variable $c_2$ with type $\texttt{T2}\langle o_2, \ldots\rangle$ can be assigned to a variable $c_1$ with type $\texttt{T1}\langle o_1, \ldots\rangle$ (either via assignment statement or passing arguments for method calls and such) if and only if $o_1 = o_2$ and $o_1 \rhd o_2$.

The detailed type rules for our type system are described in Appendix B.

**THEOREM 3.** *Our type system guarantees the following properties.*

*1. An Xmodule A can access an object with ownership tag o only if $A \preceq o$.*
*2. An Xmodule A with ownership tag $o_1$ can access another Xmodule B with ownership tag $o_2$ only if A owns B, or if $o_1 \rhd o_2$.*

PROOF. Condition 1 is the same as Boyapati et. al's access rules. Since our type system makes the type rules stricter, it still holds with our type system. [2]

Jim: Condition 1 is essentially

Condition 2 requires more explanation. An Xmodule $A$ can access another Xmodule $B$ only if inside $A$'s class file, it is possible to declare a variable x of type T and assign $B$ to $x$. The only ownership tags that $A$'s class file can use as the owner for T are one of $A$'s formal tags, or this[i] tag.

If the owner of T is one of the formal tags $o_j$, then by Property 3, we know $o_1 \rhd o_j$. By Property 4, we know $B$ can be assigned to x only if $o_j = o_2$, and $o_j \rhd o_2$. Since the relation $\rhd$ is transitive, we have $o_1 \rhd o_2$.

Similarly, if x is declared with a tag this[i], then by Property 4, we can assign $B$ to x only if $o_2 =$ this[j] (where i $\leq$ j). Thus, we have $A$ owns $B$.

□

These properties translate to the definition Definition 1 if all the children of a particular Xmodule have unique indices. By indexing this owner tags, we are able to enforce some ordering constraints between sibling Xmodules. One should note, however, that our type system can not prevent cyclic dependencies between Xmodules, since the programmer can always declare two Xmodules $A$ and $B$ with the same indexed owner this[i]. In this case, the type system does not enforce any ordering constraint between $A$ and $B$ statically. In general, it seems difficult enforce the ordering of children entirely statically (Rule 2 of Definition 1) without imposing too many programming restrictions. The runtime system, however, could dynamically check for cycles and throw a runtime error if a cycle is detected.

[2] Note that in this paper, we do not consider the possibility of inner classes, unlike the original ownership type system of [2].

## 4. OWNERSHIP-AWARE TRANSACTIONS

In this section, we informally sketch the *OAT* model, an abstract execution model for TM with ownership and Xmodules. The novel feature of the *OAT* model is that it uses the structure of Xmodules to provide a commit mechanism which can be viewed as a hybrid of closed and open nested commits. The *OAT* model presents an operational semantics for TM, and is not intended to describe an actual implementation.

***Overview***

The TM system is modeled as a nondeterministic state machine with two components: a *program* and a *runtime system*. The runtime system, which we call the *OAT* model, dynamically constructs and traverses a computation tree $\mathcal{C}$ as it executes instructions generated by the program. The *OAT* model maintains a set of ***ready*** nodes, denoted by $\texttt{ready}(\mathcal{C}) \subseteq \texttt{nodes}(\mathcal{C})$, and at every step, the *OAT* model nondeterministically chooses one of these ready nodes $X \in \texttt{ready}(\mathcal{C})$ to issue the next instruction. The program then issues one of the following instructions (whose precondition is satisfied) on $X$'s behalf: `fork`, `join`, `xbegin`, `xend`, `xabort`, `read`, or `write`. For shorthand, we sometimes say that $X$ issues an instruction.

The *OAT* model describes a sequential semantics, that is, we assume at every time step $t$, a program issues a single instruction. The parallelism in this model arises from the fact that at a particular time, several nodes can be ready, and the runtime nondeterministically chooses which one to have issue an instruction.

In the rest of this section, we give a detailed description of the *OAT* model. First, we describe the state information maintained by the *OAT* model and define the notation we use to refer to this state. Second, we describe how the *OAT* model constructs and traverses the computation tree as instructions are issued. Then, we describe how the *OAT* model handles memory operations (i.e., `read` and `write`), conflict detection, and transaction commits, and transaction aborts.

### 4.1 State Information and Notation

As the *OAT* model executes instructions, it dynamically constructs the computation tree $\mathcal{C}$. For each of the sets defined in Section 2 (e.g., $\texttt{nodes}(\mathcal{C})$, $\texttt{spNodes}(\mathcal{C})$, $\texttt{memOps}(\mathcal{C})$, $\texttt{xactions}(\mathcal{C})$, etc.), we define corresponding time-dependent versions of these sets by indexing them with an additional time argument. For example, we define the set $\texttt{nodes}(t,\mathcal{C})$ denotes the set of nodes in the computation tree after $t$ time steps have passed. The generalized sets from Section 2 are monotonically increasing, i.e., once an element is added to the set, it is never removed at a later time $t$. Sometimes for shorthand, we omit the time argument when it is clear that we are referring to a particular fixed time $t$.

Since the *OAT* model has a computation tree $\mathcal{C}$ which is dynamic, at any fixed time $t$, each internal node $A \in \texttt{spNodes}(t,\mathcal{C})$ has a ***status*** field $\texttt{status}[A]$. If $A \in \texttt{xactions}(t,\mathcal{C})$, i.e., $A$ is a transaction, then $\texttt{status}[A]$ can be one of `COMMITTED`, `ABORTED`, `PENDING`, or `PENDING_ABORT`. Otherwise, $A \in \texttt{spNodes}(t,\mathcal{C}) - \texttt{xactions}(t,\mathcal{C})$ is either a P-node or a nontransactional S-node; in this case, $\texttt{status}[A]$ can either be `WORKING` or `SYNCHED`. We define several abstract sets for the tree based on this status field. The first 6 sets partition the $\texttt{spNodes}(t,\mathcal{C})$, the set of internal nodes of the computation tree. The last 4 sets categorize transactions and nodes as being either active or complete.

1. $\texttt{pending}(t,\mathcal{C}) = \{X \in \texttt{xactions}(t,\mathcal{C}) \ : \ \texttt{status}[Z] = \texttt{PENDING}\}$ (Pending transactions).
2. $\texttt{pendingAbort}(t,\mathcal{C}) = \{X \in \texttt{xactions}(t,\mathcal{C}) \ : \ \texttt{status}[Z] = \texttt{PENDING\_ABORT}\}$ (Aborting transactions).
3. $\texttt{committed}(t,\mathcal{C}) = \{X \in \texttt{xactions}(t,\mathcal{C}) \ : \ \texttt{status}[Z] = \texttt{COMMITTED}\}$ (Committed transactions).
4. $\texttt{aborted}(t,\mathcal{C}) = \{X \in \texttt{xactions}(t,\mathcal{C}) \ : \ \texttt{status}[Z] = \texttt{ABORTED}\}$ (Aborted transactions).
5. $\texttt{working}(t,\mathcal{C}) = \{Z \in \texttt{spNodes}(t,\mathcal{C}) - \texttt{xactions}(t,\mathcal{C}) \ : \ \texttt{status}[Z] = \texttt{WORKING}\}$ (Working nodes).
6. $\texttt{synched}(t,\mathcal{C}) = \{Z \in \texttt{spNodes}(t,\mathcal{C}) - \texttt{xactions}(t,\mathcal{C}) \ : \ \texttt{status}[Z] = \texttt{SYNCHED}\}$ (Synched nodes).
7. $\texttt{activeX}(t,\mathcal{C}) = \texttt{pending}(t,\mathcal{C}) \cup \texttt{pendingAbort}(t,\mathcal{C})$ (Active transactions).
8. $\texttt{activeN}(t,\mathcal{C}) = \texttt{activeX}(t,\mathcal{C}) \cup \texttt{working}(t,\mathcal{C})$. (Active nodes).

9. $\texttt{doneX}(t,\mathcal{C}) = \texttt{committed}(t,\mathcal{C}) \cup \texttt{aborted}(t,\mathcal{C})$ (Complete transactions).
10. $\texttt{doneN}(t,\mathcal{C}) = \texttt{doneX}(t,\mathcal{C}) \cup \texttt{synched}(t,\mathcal{C})$ (Complete nodes).

The *OAT* model maintains a set of ***ready*** S-nodes, denoted as $\texttt{ready}(t,\mathcal{C})$. We discuss the properties of ready nodes later, in Section 4.2. Note that $\texttt{ready}(t,\mathcal{C})$, and the sets defined above which are subsets of $\texttt{activeN}(t,\mathcal{C})$ are not monotonic, because completing nodes removes elements from these sets.

For the purposes of detecting conflicts, at any time $t$, for any active transaction $T$, i.e., $T \in \texttt{activeX}(t,\mathcal{C})$, the *OAT* model maintains a ***readset*** $\texttt{R}(t,T)$ and a ***writeset*** $\texttt{W}(t,T)$ for $T$. The readset $\texttt{R}(t,T)$ is a set of pairs $(\ell,v)$, where $\ell \in \mathcal{L}$ is a memory location and $v \in \texttt{memOps}(t,\mathcal{C})$ is a memory operation that reads from $\ell$. We define $\texttt{W}(t,T)$ similarly. We represent main memory as the readset/writeset of $\texttt{root}(\mathcal{C})$. At time $t = 0$, we assume $\texttt{R}(0,\texttt{root}(\mathcal{C}))$ and $\texttt{W}(0,\texttt{root}(\mathcal{C}))$ initially contain a pair $(\ell,\perp)$ for all locations $\ell \in \mathcal{L}$.

The *OAT* model maintains two invariants on $\texttt{R}(t,T)$ and $\texttt{W}(t,T)$. First, $\texttt{W}(t,T) \subseteq \texttt{R}(t,T)$ for every transaction $T \in \texttt{xactions}(t,\mathcal{C})$, i.e., a write also counts as a read. Second, $\texttt{R}(t,T)$ and $\texttt{W}(t,T)$ each contain at most one pair $(\ell,v)$ for any location $\ell$. Thus, we use the shorthand $\ell \in \texttt{R}(t,T)$ to mean that there exists a node $u$ such that $(\ell,u) \in \texttt{R}(t,T)$, and similarly for $\texttt{W}(t,T)$. We also overload the union operator: at some time $t$, an operation $\texttt{R}(T) \leftarrow \texttt{R}(T) \cup \{(\ell,u)\}$ means we construct $\texttt{R}(t+1,T)$ by

$$\texttt{R}(t+1,T) = \{(\ell,u)\} \cup \left(\texttt{R}(t,T) - \left\{(\ell,u') \in \texttt{R}(t,T)\right\}\right).$$

In other words, we add $(\ell,u)$ to $\texttt{R}(T)$, replacing any $(\ell,u') \in \texttt{R}(t,T)$ that existed previously.

Finally, for a transaction $T \in \texttt{activeX}(t,\mathcal{C})$, we also define a ***module readset*** as

$$\texttt{modR}(t,T) = \{(\ell,v) \in \texttt{R}(t,T) \ : \ \texttt{owner}(\ell) = \texttt{xMod}(T)\}.$$

In other words, $\texttt{modR}(t,T)$ is the subset of $\texttt{R}(t,T)$ that accesses memory owned by $T$'s Xmodule $\texttt{xMod}(T)$. Similarly, we define the ***module writeset*** as

$$\texttt{modW}(t,T) = \{(\ell,v) \in \texttt{W}(t,T) \ : \ \texttt{owner}(\ell) = \texttt{xMod}(T)\}.$$

### 4.2 Constructing the Computation Tree

In the *OAT* model, the runtime constructs the computation tree in a straightforward fashion as instructions are issued. The *OAT* model maintains a computation tree that satisfies two structural properties.

First, the *OAT* model builds only computation trees $\mathcal{C}$ which have the following canonical form.

**PROPERTY 1.** *A canonical computation tree $\mathcal{C}$ satisfies the following properties.*

1. $\texttt{root}(\mathcal{C})$ *is a transaction.*
2. *All transactions* $Z \in \texttt{xactions}(\mathcal{C})$ *are S-nodes.*
3. *In $\mathcal{C}$, every P-node $Y$ has exactly two nontransactional S-nodes $Z_1$ and $Z_2$ as children, and* $\texttt{parent}(Y)$ *is an S-node.*

Second, at any time $t$, if one looks only the active nodes $\texttt{activeN}(t,\mathcal{C})$, the *OAT* model maintains the invariant the active nodes form a tree, with the ready nodes at the leaves. In other words, the *OAT* model preserves the following invariant.

**PROPERTY 2.** *At any time $t$, the computation tree $\mathcal{C}$ satisfies these properties:*

1. *For all* $X \in \texttt{ready}(t,\mathcal{C})$, $\texttt{ances}(X) \subseteq \texttt{activeN}(t,\mathcal{C})$.
2. *For all* $X \in \texttt{ready}(t,\mathcal{C})$, $(\texttt{pDesc}(X) \cap \texttt{nodes}(t,\mathcal{C})) \subseteq \texttt{doneN}(t,\mathcal{C})$.

*In other words, the set* $\texttt{activeN}(t,\mathcal{C})$ *forms an* ***active tree****.*

Since the *OAT* model is a sequential semantics, it is clear that the sequence of instructions always generates a valid topological sort $\mathcal{S}$ of the computation dag, $G(\mathcal{C})$.

Jim: Don't know where this s

The instructions in the *OAT* model maintain Properties 1 and 2 for the computation tree in a straightforward fashion. For completeness, however, we give a more detailed description of this construction.

Initially, at time $t = 0$, we begin with only the root node in the tree, i.e., $\texttt{nodes}(0, \mathcal{C}) = \texttt{xactions}(0, \mathcal{C}) = \{\texttt{root}(\mathcal{C})\}$. Throughout the entire computation, the *OAT* model always maintains $\texttt{status}[\texttt{root}(\mathcal{C})] = \texttt{PENDING}$, i.e., the root node of the tree is always PENDING. This root node also begins as ready, i.e., $\texttt{ready}(0, \mathcal{C}) = \{\texttt{root}(\mathcal{C})\}$.

The *OAT* model creates new internal nodes in $\mathcal{C}$ during time step $t+1$ when it chooses a ready node $X \in \texttt{ready}(t, \mathcal{C})$ and has $X$ issue a fork or xbegin instruction. If $X$ issues a fork, then the runtime creates a P-node $P$ as a child of $X$, and two S-nodes $S_1$ and $S_2$ as children of $P$, all with status WORKING. The fork also removes $X$ from $\texttt{ready}(\mathcal{C})$ and adds $S_1$ and $S_2$ to $\texttt{ready}(\mathcal{C})$. If $X$ issues an xbegin, then the runtime creates a new transaction $Y \in \texttt{xactions}(\mathcal{C})$ as a child of $X$, with $\texttt{status}[Y] = \texttt{PENDING}$, removes $X$ from $\texttt{ready}(\mathcal{C})$, and adds $Y$ to $\texttt{ready}(\mathcal{C})$.

The *OAT* model completes a nontransactional S-node $Z \in \texttt{ready}(t, \mathcal{C}) - \texttt{xactions}(t, \mathcal{C})$ (which must have $\texttt{status}[Z] = \texttt{WORKING}$) by having $Z$ issue a join instruction. The join instruction first changes $\texttt{status}[Z]$ to SYNCHED. In the tree, since $\texttt{parent}(Z)$ is always a P-node, $Z$ has exactly one sibling. If $Z$ is the first child of $\texttt{parent}(Z)$ to be SYNCHED, the *OAT* model removes $Z$ from $\texttt{ready}(\mathcal{C})$. Otherwise, $Z$ is the last child of $\texttt{parent}(Z)$ to be SYNCHED, and the *OAT* model removes $Z$ and $\texttt{parent}(Z)$ from $\texttt{ready}(\mathcal{C})$ and adds $\texttt{parent}(\texttt{parent}(Z))$ to $\texttt{ready}(\mathcal{C})$.

oes this actually make any
to be careful about having

The *OAT* model can complete a transaction $X \in \texttt{ready}(t, \mathcal{C})$ by having it issue either an xend or xabort instruction. If $\texttt{status}[X] = \texttt{PENDING}$, then $X$ can issue an xend to change $\texttt{status}[X]$ to COMMITTED. Otherwise, $\texttt{status}[X] = \texttt{PENDING\_ABORT}$, and $X$ can issue an xabort to change its status to ABORTED. For both xend and xabort, the *OAT* model removes $X$ from $\texttt{ready}(\mathcal{C})$ and adds $\texttt{parent}(X)$ back into $\texttt{ready}(\mathcal{C})$. The xend instruction also performs an ownership-aware commit and changes readsets and writesets, which we describe later in Section 4.4.

Finally, a ready node $X$ issues a read and write instruction, if the instruction does not generate a conflict, it adds a memory operation node $v$ to $\texttt{memOps}(t, \mathcal{C})$, with $v$ as a child of $X$. If the instruction would create a conflict, the runtime may change the status of one PENDING transaction $T$ to PENDING_ABORT to make progress in resolving the conflict. For shorthand, we refer to the status change of a transaction $T$ from PENDING to PENDING_ABORT as a sigabort of $T$.

### 4.3 Memory Operations and Conflict Detection

The *OAT* model performs eager conflict detection; before performing a memory operation that would create a new $v \in \texttt{memOps}(\mathcal{C})$, the *OAT* model first checks whether creating $v$ would cause a conflict, according to Definition 3.

**DEFINITION 3.** *Suppose at time $t$, the OAT model issues a* `read` *or* `write` *instruction that potentially creates a memory operation node $v$. We say that $v$ generates a* ***memory conflict*** *if there exists a location $\ell \in \mathcal{L}$ and an active transaction $T_u \in \texttt{activeX}(t, \mathcal{C})$ such that*

1. $T_u \notin \texttt{xAnces}(v)$, *and*
2. *either* $R(v, \ell) \wedge ((\ell, u) \in \texttt{W}(t, T_u))$, *or* $W(v, \ell) \wedge ((\ell, u) \in \texttt{R}(t, T_u))$.

If $v$ would generate a conflict, then the memory operation $v$ does not occur; instead, a sigabort of some transaction may occur. We describe the mechanism for aborts in Section 4.5.

Otherwise, $v$ does not generate a conflict. Then, $v$ observes the value $\ell$ from $\texttt{R}(Y)$, where $Y$ is the closest ancestor of $v$ with $\ell$ in its readset (i.e., $(\ell, u) \in \texttt{R}(Y)$ and $\Phi(v) = u$). The read also adds $v$ to $X$'s readset.

A successful write operation $v$ sets the observer function $\Phi(v)$ in the same way as a read. The write adds $(\ell, v)$ to both $\texttt{R}(X)$ and $\texttt{W}(X)$.

### 4.4 Ownership-Aware Transaction Commit

The *OAT* model implements an ownership-aware commit mechanism for nested transactions which contains elements of both a closed-nested and an open-nested commit. A PENDING transaction $Y$ issues an xend instruction to commit $Y$ into $X = \texttt{xparent}(Y)$. When $Y$ commits, it commits locations from its readset/writeset which are owned by $\texttt{xMod}(Y)$'s in an open-nested fashion to the root of the tree, while it commits locations owned by other Xmodules in a closed-nested fashion, by propagating those reads/writes to $X$.

We can describe the *OAT* model's commit mechanism more formally in terms of module readsets and writesets. Suppose at time $t$, $Y \in \texttt{xactions}(t, \mathcal{C})$ with $\texttt{status}[Y] = \texttt{PENDING}$ issues an xend. This xend changes readsets and writesets as follows.

$$\begin{aligned}
\texttt{R}(\texttt{root}(\mathcal{C})) &\leftarrow \texttt{R}(\texttt{root}(\mathcal{C})) \cup \texttt{modR}(Y) \\
\texttt{R}(\texttt{xparent}(Y)) &\leftarrow \texttt{R}(\texttt{xparent}(Y)) \cup (\texttt{R}(Y) - \texttt{modR}(Y)) \\
\texttt{W}(\texttt{root}(\mathcal{C})) &\leftarrow \texttt{W}(\texttt{root}(\mathcal{C})) \cup \texttt{modW}(Y) \\
\texttt{W}(\texttt{xparent}(Y)) &\leftarrow \texttt{W}(\texttt{xparent}(Y)) \cup (\texttt{W}(Y) - \texttt{modW}(Y))
\end{aligned}$$

For a memory operation $u$, Theorem 2 implies that the the ownership-aware commit mechanism has a well-defined "committer" for $u$.

**DEFINITION 4.** *For any memory operation $u$, which accesses a location $\ell$, define the* committer *of $u$, denoted* $\texttt{committer}(u)$*, as the unique transaction $T^*$ from Theorem 2 such that* $\texttt{owner}(\ell) = \texttt{xMod}(T^*)$.

Intuitively, $\texttt{committer}(u)$ is the transaction which "belongs" to the same Xmodule as the location $\ell$ which $u$ accesses, and is responsible for committing $u$ to memory. One can also show for any $u$ which accesses a location $\ell$, $\ell$ can never appear in the readset (or writeset) of any transaction $T'$ which is an ancestor of $\texttt{committer}(u)$. Note that this property does not hold for TM with open-nested commits; in that case, $\texttt{R}(T')$ may contain a different value for $\ell$ that may be replaced upon commit. Jim: THIS WAS AN OLD PAR

For programs where every Xmodule $A$ accesses only locations $\ell$ which it owns, an open-nested commit is equivalent to an ownership-aware commit because any memory modified by $T$ with $\texttt{xMod}(T) = A$ is committed directly to $\texttt{root}(\mathcal{C})$. Some program examples, however, are arguably easier to reason about using an ownership-aware commit. For instance, suppose in the example application from Section 1, that a Book object has a field of lastSearched that keeps track of the last time a query was performed involving that Book in a successful top-level transaction. Suppose this field is also read by the UserApp Xmodule. In this case, if the BST uses an open-nested commit, the programmer must worry about not only the commutativity with methods in BST Xmodule, but also the commutativity with methods in the UserApp Xmodule that access (read or write) the lastSearched field. Similarly, when compensating the methods of the BST Xmodule, the compensating action would need to undo the modification to the lastSearched field. With an ownership-aware commit mechanism, on the other hand, the write on the lastSearched field is then propagated up to the parent transaction, and eventually committed to memory only when a top-level transaction of the UserApp Xmodule ends, (since we assume the Book instance is owned by the UserApp).

### 4.5 Transaction Abort

When the *OAT* model detects a conflict, it aborts one of the conflicting transactions by changing its status from PENDING to PENDING_ABORT. In the *OAT* model, a transaction $T \in \texttt{xactions}(\mathcal{C})$ might not abort immediately; instead, it might continue to issue more instructions after it's status has changed to PENDING_ABORT. This condition allows the system to use compensating actions to compensate for the nested transactions that may have committed. Eventually a PENDING_ABORT transaction issues an xend instruction, which then changes its status from PENDING_ABORT to ABORTED.

Later, it will be useful to refer to the set of operations a transaction $T$ issues while its status is PENDING_ABORT.

**DEFINITION 5.** *The set of operations issued by T or its descendants after T's status changes to* PENDING_ABORT *are called T's* ***abort actions*** *. This set is denoted by* abortactions$(T)$.

If a potential memory operation *v* generates a conflict with $T_u$ and $T_u$'s status is PENDING, then the *OAT* model can nondeterministically choose to abort either xparent$(v)$, or $T_u$. In the latter case, *v* then "waits" for $T_u$ to finish aborting (i.e., change its status to ABORTED) before continuing. If $T_u$'s status is PENDING_ABORT, then *v* just waits for $T_u$ to finish aborting before trying to issue read or write again. [3]

This operational model uses the same conflict detection algorithm as TM with ordinary closed-nested transactions does; the only subtleties are that *v* can generate a conflict with a PENDING_ABORT transaction $T_u$, and that transactions no longer abort instantaneously because they have abort actions. Some restrictions on the abort actions of a transaction may be necessary to avoid deadlock, as we describe later in Section 6.

## 5. SERIALIZABILITY BY MODULES

In this section, we define ***serializability by modules***, a definition inspired by the database definition of multilevel serializability (e.g., as described in [11]). We then provide a proof sketch that the *OAT* model from Section 4 guarantees serializability by modules.

First, we describe the definition of serializability in the transactional computation framework, as given in [1]. Next, we incorporate Xmodules into this definition and define serializability by modules. We then prove that the *OAT* model guarantees serializability by modules. Finally, we discuss the relationship between the definition of serializability by modules, and the notion of abstract serializability for the methodology of open nesting.

### 5.1 Transactional Computations and Serializability

In [1], serializability for a transactional computation with computation tree $\mathcal{C}$ was defined in terms of topological sorts $\mathcal{S}$ of the computation dag $G(\mathcal{C})$. Informally, a trace $(\mathcal{C},\Phi)$ is serializable if there exists a topological sort order $\mathcal{S}$ of $G(\mathcal{C})$ such that $\mathcal{S}$ is "sequentially consistent with respect to $\Phi$", and all transactions appear contiguous in the order $\mathcal{S}$. In this section, we give more precise, formal definitions of this concept.

#### *Content Sets*

For a given trace $(\mathcal{C},\Phi)$, we define "content" sets for every transaction $T$ by partitioning memOps$(T)$ into three sets: cContent$(T)$, oContent$(T)$ and aContent$(T)$. For any $u \in$ memOps$(T)$, we define the content sets based on the status of transactions in $\mathcal{C}$ that one visits when walking up the tree from *u* to $T$.

**DEFINITION 6.** *For any transaction T and memory operation u, define the sets* cContent$(T)$, oContent$(T)$, *and* aContent$(T)$ *according the* ContentType$(u,T)$ *procedure:*

```
    ContentType(u,T)          ▷ For any u ∈ memOps(T)
1   X ← xparent(u)
2   while (X ≠ T)
3       if (X is ABORTED)        return u ∈ aContent(T)
4       if (X = committer(u))    return u ∈ oContent(T)
5       X ← xparent(X)
6   return u ∈ cContent(T)
```

Recall that in the *OAT* model, the commit of $T$ commits some memory operations in an open-nested fashion, directly to memory, and some operations in a closed-nested fashion, to parent$(T)$. Informally,

[3] If *v* causes a conflict, we know that $Z =$ parent$(v)$ and $Z \in$ ready$(\mathcal{C})$; waiting until $T_u$ has finished aborting can be modeled as either the runtime not choosing $Z$ as a ready node to issue an instruction until an xabort for $T_u$ occurs, or having $Z$ issue "nop" instructions until $T_u$ as finished aborting.

$\texttt{oContent}(T)$ is the set of memory operations that are committed in an "open" manner by $T$'s subtransactions. Similarly, $\texttt{aContent}(T)$ is the set of operations that are discarded due to the abort of some subtransaction in $T'$ subtree. Finally, $\texttt{cContent}(T)$ is the set of operations that are neither committed in an "open" manner, nor aborted.

***Sequential Consistency with Transactions***

For computations with transactions, we can modify the classic notion of sequential consistency to account for transactions which abort. Transactional semantics dictate that memory operations belonging to an aborted transaction $T$ should not be observed by (i.e., ***hidden*** from) memory operations outside of $T$.

**DEFINITION 7.** *For any two vertices $u, v \in V(\mathcal{C})$, let $X = \texttt{xLCA}(u,v)$. We say that $u$ is **hidden** from $v$, denoted $uHv$, if $u \in \texttt{aContent}(X)$.*

Our definition of serializability by modules requires that computations satisfy some notion of sequential consistency, generalized for the setting of TM.

**DEFINITION 8.** *Consider a trace $(\mathcal{C},\Phi)$ and a topological sort $\mathcal{S}$ of $G(\mathcal{C})$. For all $v \in \texttt{memOps}(\mathcal{C})$ such that $R(v,\ell) \vee W(v,\ell)$, the **transactional last writer** of $v$ according to $\mathcal{S}$, denoted $\mathcal{X}_{\mathcal{S}}(v)$, is the unique $u \in \texttt{memOps}(\mathcal{C}) \cup \{\perp\}$ that satisfies four conditions:*

1. *$W(u,\ell)$,*
2. *$u <_{\mathcal{S}} v$,*
3. *$\neg(uHv)$, and*
4. *$\forall w\,(W(w,\ell) \wedge (u <_{\mathcal{S}} w <_{\mathcal{S}} v)) \Rightarrow wHv$.*

**DEFINITION 9.** *A trace $(\mathcal{C},\Phi)$ is **sequentially consistent** if there exists a topological sort $\mathcal{S}$ such that $\Phi = \mathcal{X}_{\mathcal{S}}$. We say that $\mathcal{S}$ is **sequentially consistent with respect to** $\Phi$.*

In other words, the transactional last writer of a memory operation $u$ which accesses location $\ell$, is the last write $v$ to location $\ell$ in the order $\mathcal{S}$, except we skip over writes $w$ which are hidden from (i.e., aborted with respect to) $u$. Intuitively, Definition 9 requires that there exists an order $\mathcal{S}$ explaining all the memory operations of the computation.

***Serializability***

**DEFINITION 10.** *A trace $(\mathcal{C},\Phi)$ is **serializable** if there exists a topological sort $\mathcal{S}$ that satisfies two conditions:*

1. *$\Phi = \mathcal{X}_{\mathcal{S}}$ ($\mathcal{S}$ is sequentially consistent with respect to $\Phi$), and*
2. *$\forall T \in \texttt{xactions}(\mathcal{C})$ and $\forall v \in V(\mathcal{C})$, we have $\texttt{xbegin}(T) \leq_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{xend}(T)$ implies $v \in V(T)$).*

Ordinary serializability can be thought of as a strengthening of sequential consistency which also requires that the order $\mathcal{S}$ both explains all memory operations, and also has all transactions appearing contiguous.

### 5.2 Defining Serializability by Modules

In [1], a trace $(\mathcal{C},\Phi)$ was said to be *serializable* if there exists a topological sort $\mathcal{S}$ of $G(\mathcal{C})$ such that $\mathcal{S}$ is sequentially consistent with respect to $\Phi$, and all transactions appear contiguous in $\mathcal{S}$. Serializability in this context can be thought of as a sequential consistency plus the requirement that transactions are atomic. For ownership-aware transactions, this definition of serializability is too strong because conflicting accesses to memory owned by a low-level Xmodule causes transactions of a higher-level Xmodule to conflict, preventing these transactions from commuting with each other.

Instead, we describe a definition of serializability by modules which checks for correctness one Xmodule at a time. Informally, the definition proceeds as follows. Given a trace $(\mathcal{C},\Phi)$, for each Xmodule $A$, we

transform the tree $\mathcal{C}$ into a new tree $\texttt{mTree}(\mathcal{C},A)$, and then check that in the trace $(\texttt{mTree}(\mathcal{C},A),\Phi)$, that only the transactions of Xmodule $A$ are serializable. The new tree $\texttt{mTree}(\mathcal{C},A)$ is constructed in such a way as to ignore memory operations of Xmodules which are lower-level than $A$, and also to ignore all operations which are hidden from transactions of $A$. If the check holds for all Xmodules, then trace $(\mathcal{C},\Phi)$ is said to be serializable by modules. We construct $\texttt{mTree}(\mathcal{C},A)$ according to Definition 11.

**DEFINITION 11.** *For any computation tree $\mathcal{C}$, let $\texttt{mTree}(\mathcal{C},A)$ be the result of modifying $\mathcal{C}$ as follows:*

1. *For all memory operations $u \in \texttt{memOps}(\mathcal{C})$ with $u$ accessing $\ell$, if $\texttt{owner}(\ell) = B$ for some $\texttt{level}(B) < \texttt{level}(A)$, convert $u$ into a nop.*
2. *For all transactions $T \in \texttt{modXactions}(A)$, convert all $u \in \texttt{aContent}(T)$ into nops.*

The intuition behind Step 1 of Definition 11 is that when looking at Xmodule $A$, we throw away memory operations belonging to a lower-level Xmodule $B$, since by Theorem 2, transactions of $A$ can never directly access the same memory as those operations anyway. For Step 2, we ignore the content of any aborted transactions nested inside transactions of $A$; those transactions might access the same memory locations as operations which we did not turn into nops, but those operations are aborted with respect to transactions of $A$.

Lemma 4 argues that for a trace which is originally sequentially consistent, turning memory operations into nops according to Definition 11 does not create an invalid trace, i.e., one where an operation $u$ that remains in the trace attempts to observe a value from a $\Phi(u)$ which was turned into a nop.

**LEMMA 4.** *Let $(\mathcal{C},\Phi)$ be any sequentially consistent trace. Then for any Xmodule $A$, $(\texttt{mTree}(\mathcal{C},A),\Phi)$ is a valid trace. In other words, if $u \in \texttt{memOps}(\texttt{mTree}(\mathcal{C},A))$, then $\Phi(u) \in \texttt{memOps}(\texttt{mTree}(\mathcal{C},A))$). Furthermore, any $\mathcal{S}$ which is sequentially consistent for $\Phi$ in $(\mathcal{C},\Phi)$ is also sequentially consistent for $\Phi$ in $(\texttt{mTree}(\mathcal{C},A),\Phi)$.*

PROOF. In the new tree $\texttt{mTree}(\mathcal{C},A)$, pick any $u \in \texttt{memOps}(\texttt{mTree}(\mathcal{C},A))$ which remains. Assume for contradiction that $v = \Phi(u)$ was turned into a nop in one of Steps 1 and 2.

If $v$ was turned into a nop in Step 1, the we know because $v$ accessed an $\ell$ satisfying $\texttt{level}(\texttt{owner}(\ell)) < \texttt{level}(A)$. Since $u$ must access the same location $\ell$, $u$ must also be converted into a nop.

If $v$ was turned into a nop in Step 2, then $v \in \texttt{aContent}(T)$ for some $\texttt{xMod}(T) = A$. Then we can show that either $vHu$, or $u$ should have also been turned into a nop. Let $X = \texttt{xLCA}(v,u)$. Since $X$ and $T$ are both ancestors of $v$, either $X$ is an ancestor of $T$ or $T$ is a proper ancestor of $X$.

1. First, suppose $T$ is a proper ancestor of $X$. Consider the path of transactions $Y_0, Y_1, \ldots Y_k$, where $Y_0 = \texttt{xparent}(v)$, $\texttt{xparent}(Y_i) = Y_{i+1}$, and $\texttt{xparent}(Y_k) = T$. Since $v \in \texttt{aContent}(T)$, for some $Y_j$ for $0 \le j \le k$ must have $\texttt{status}[Y_j] = \texttt{ABORTED}$. Since $T$ is a proper ancestor of $X$, $X = Y_x$ for some $x$ satisfying $0 \le x \le k$.
   (a) If $\texttt{status}[Y_j] = \texttt{ABORTED}$ for any $j$ satisfying $0 \le j < x$, then we know $v \in \texttt{aContent}(X)$, and thus $vHu$. Since we assumed $(\mathcal{C},\Phi)$ is sequentially consistent and $\Phi(v) = u$, by Definition 8, we know $\neg vHu$, leading to a contradiction.
   (b) If $Y_j$ is $\texttt{ABORTED}$ for any $j$ satisfying $x \le j \le k$, then $\texttt{status}[Y_j] = \texttt{ABORTED}$ implies that $u \in \texttt{aContent}(X)$, and thus, $u$ should have been turned into a nop, contradicting the original setup of the statement.
2. Next, consider the case where $X$ is an ancestor of $T$. Since $v \in \texttt{aContent}(T)$, we have $v \in \texttt{aContent}(X)$. Therefore, this case is analogous to Case 1a above.

Finally, if $\Phi$ is the transactional last writer according to $\mathcal{S}$ for $(\mathcal{C},\Phi)$, it is still the transactional last writer for $(\texttt{mTree}(\mathcal{C},A),\Phi)$ because the memory operations which are not turned into nops remain in the same relative order. Thus, the last condition is satisfied.

□

Note that Lemma 4 *depends on* the restrictions described in Definition 1. Without this structure of modules and ownership, the construction of Definition 11 is not guaranteed to generate a valid trace. Also, note that the set of memory operations which are turned into nops strictly increases as we look at $\texttt{mTree}(\mathcal{C},A)$ and increase $\texttt{level}(A)$. For the lowest-level Xmodule, say $A_0$, we keep all memory operations (i.e., $\texttt{mTree}(\mathcal{C},A_0) = \mathcal{C}$). Once a memory operation *u* is turned into a nop for Xmodule *A*, it is turned into a nop for all Xmodules *B* with $\texttt{level}(B) > \texttt{level}(A)$.

Finally, we can define serializability by modules.

**DEFINITION 12.** *A trace $(\mathcal{C},\Phi)$ is **serializable by modules** if it is sequentially consistent, and if for all Xmodules A in $\mathcal{D}$, there exists a topological sort $\mathcal{S}$ of $\mathcal{C}_A = \texttt{mTree}(\mathcal{C},A)$ such that:*

1. *$\Phi = \mathcal{X}_\mathcal{S}$, ($\mathcal{S}$ is sequentially consistent with respect to $\Phi$), and*
2. *For the tree $\mathcal{C}_A$, $\forall T \in \texttt{modXactions}(A)$ and $\forall v \in V(\mathcal{C}_A)$, if we have $\texttt{xbegin}(T) \leq_\mathcal{S} v \leq_\mathcal{S} \texttt{xend}(T)$, then $v \in V(T)$.*

Informally, a trace $(\mathcal{C},\Phi)$ is serializable by modules if it is sequentially consistent, and if for every Xmodule *A*, there exists a sequentially consistent order $\mathcal{S}$ for the trace $(\texttt{mTree}(\mathcal{C},A),\Phi)$ which also has all transactions of *A* contiguous.

### 5.3 *OAT* Model Guarantees Serializability by Modules

In this section, we show that the *OAT* model described in Section 4 generates traces $(\mathcal{C},\Phi)$ that are serializable by modules, i.e., that satisfy Definition 12. The proof of this fact consists of three steps. First, we generalize the notion of "prefix race-freedom" described in [1], to computations with Xmodules. Second, we prove that the *OAT* model guarantees that a program execution is prefix race-free. Finally, we argue that any trace which is prefix race-free is also serializable by modules.

#### *Defining Prefix Race-Freedom*

First, we define the prefix races. These definitions are essentially the same as those in [1], except adapted for a system with an ownership-aware commit mechanism instead of an open-nested commit mechanism.

**DEFINITION 13.** *For any execution order $\mathcal{S}$, for any transaction $T \in \texttt{xactions}(\mathcal{C})$, consider any $v \notin \texttt{memOps}(T)$ such that $\texttt{xbegin}(T) <_\mathcal{S} v <_\mathcal{S} \texttt{xend}(T)$, we say there exists a **prefix race** between T and v if there exists a memory operation $w \in \texttt{cContent}(T)$ s.t., $w <_\mathcal{S} v$, $\neg(vHw)$, v and w both access $\ell$, and one of v, w writes to $\ell$.*

**DEFINITION 14.** *A trace $(\mathcal{C},\Phi)$ is **prefix race-free** iff exists a topological sort $\mathcal{S}$ of $G(\mathcal{C})$ satisfying two conditions:*

1. *$\Phi = \mathcal{X}_\mathcal{S}$ ($\mathcal{S}$ is sequentially consistent with respect to $\Phi$), and*
2. *$\forall v \in V(\mathcal{C})$ and $\forall T \in \texttt{xactions}(\mathcal{C})$ there is no prefix race between v and T.*

*$\mathcal{S}$ is called a **prefix race-free sort** of the trace.*

#### *Properties of the OAT Model*

Second, we prove several invariants that *OAT* model preserves, and then use these invariants to prove that the *OAT* model generates only traces $(\mathcal{C},\Phi)$ which are prefix race-free.

The sequence of instructions that the *OAT* model issues naturally generates a topological sort $\mathcal{S}$ of the computation dag $G(\mathcal{C})$: the `fork` and `xbegin` instructions correspond to the begin nodes of a parallel or

series blocks in the dag, the join, xend, and xabort instructions correspond to end nodes of parallel or series blocks, and the read or write instructions correspond to memory operation nodes $v \in \texttt{memOps}(\mathcal{C})$.

**THEOREM 5.** *Suppose the OAT model generates a trace $(\mathcal{C}, \Phi)$ and an execution order $\mathcal{S}$. Then, $\Phi = \mathcal{X}_{\mathcal{S}}$, i.e., $\mathcal{S}$ is sequentially consistent with respect to $\Phi$.*

PROOF. This result is reasonably intuitive, but the proof is tedious and somewhat complicated. We defer the details of this proof to Appendix A. □

Next, we describe an invariant on readsets and writesets that the *OAT* model maintains.

**LEMMA 6.** *Suppose the OAT model generates a trace $(\mathcal{C}, \Phi)$ with an execution order $\mathcal{S}$. For any transaction $T$, consider a memory operation $u \in \texttt{cContent}(T)$ which accesses memory location $\ell$ at step $t_0$. Let $t_f$ be step when $\texttt{xend}(T)$ or $\texttt{xabort}(T)$ happens. At any time $t$ such that $t_0 \leq t < t_f$ there exists some active transaction $T' \in \texttt{xDesc}(T) \cap \texttt{activeX}(t, \mathcal{C})$ (which is a descendant of $T$) such that*

1. *If $R(u, \ell)$, then $\ell \in \texttt{R}(t, T')$.*
2. *If $W(u, \ell)$, then $\ell \in \texttt{W}(t, T')$.*

PROOF. Let $X_1, X_2, \ldots X_k$ be the chain of transactions from $\texttt{xparent}(u)$ up to, but not including $T$, i.e., $X_1 = \texttt{xparent}(u)$, $X_j = \texttt{xparent}(X_{j-1})$, and $\texttt{xparent}(X_k) = T$. Since we assume $u \in \texttt{cContent}(T)$, and since $T$ completes at time $t_f$, we know at some time $t_j$ which satisfies $t_0 \leq t_j < t_f$, an xend changes $\texttt{status}[X_j]$ from PENDING to COMMITTED; otherwise, we would have $u \in \texttt{aContent}(T)$.

Also, by Definitions 4 and 6, we know $\texttt{committer}(u) \in \texttt{xAnces}(T)$, i.e., none of the $X_j$'s will commit location $\ell$ in an open-nested fashion to the world; otherwise, we would have $u \in \texttt{oContent}(T)$.

First, suppose $R(u, \ell)$. At time $t_i$, when the memory operation $u$ completes, $(\ell, u)$ is added to $\texttt{R}(X_1)$. In general, at time $t_j$, the ownership-aware commit mechanism, as described in Section 4.4, will propagate $\ell$ from $\texttt{R}(X_j)$ to $\texttt{R}(X_{j+1})$. Therefore, for any time $t$ in the interval $[t_{j-1}, t_j)$, we know $\ell \in \texttt{R}(t, X_j)$, i.e., for Lemma 6, $T' = X_j$. Similarly, for any time $t$ in the interval $[t_k, t_f)$, we have $\ell \in \texttt{R}(t, T)$, i.e., we choose $T' = T$.

The case where $W(u, \ell)$ is completely analogous to the case of $R(u, \ell)$, except we have both $\ell \in \texttt{R}(t, T')$ and $\ell \in \texttt{W}(t, T')$. □

Informally, Lemma 6 states that, if a memory operation $u$ that reads / writes location $\ell$ is in the cContent(T) for some transaction $T$, then $l$ is pending in the readset / writeset of some active transaction under $T$'s subtree between the time when the memory operation is performed and the time when $T$ ends.

Finally, we use Theorem 5 and Lemma 6 to prove that the *OAT* model generates traces which are prefix race-free.

**THEOREM 7.** *Suppose the OAT model generates a trace $(\mathcal{C}, \Phi)$ with an execution order $\mathcal{S}$. Then $\mathcal{S}$ is an prefix race-free sort of $(\mathcal{C}, \Phi)$.*

PROOF.

For the first condition of Definition 14, we know by Theorem 5, we know the *OAT* model generates an order $\mathcal{S}$ which is sequentially consistent with respect to $\Phi$.

To check the second condition, assume for contradiction that we have an order $\mathcal{S}$ generated by the *OAT* model, but there exists a prefix race between a transaction $T$ and a memory operation $v \notin \texttt{memOps}(T)$. Let $w$ be the memory operation from Definition 13, i.e., $w \in \texttt{cContent}(T)$, $w <_{\mathcal{S}} v <_{\mathcal{S}} \texttt{xend}T$, $\neg(vHw)$, $w$ and $v$ access the same location $\ell$, with one of the accesses being a write. Let $t_w$ and $t_v$ be the time steps in which operations $w$ and $v$ occurred, respectively, and let $t_{endT}$ be the time at which either $\texttt{xend}(T)$ or $\texttt{xabort}(T)$ occurs (i.e., either $T$ commits or aborts). We argue that at time $t_v$, the memory operation $v$ should not have succeeded because it generated a conflict.

We consider three cases. First suppose $W(v, \ell)$ and $R(w, \ell)$. Since $t_w < t_v < t_{endT}$, by Lemma 6, at time $t_v$, $\ell$ is in the writeset of some active transaction $T' \in \texttt{desc}(T)$. Since $v \notin \texttt{memOps}(T)$, we know $T \notin \texttt{ances}(v)$.

Thus, since $T'$ is a descendant of $T$, we have $T' \notin \texttt{ances}(v)$. Since $T' \notin \texttt{ances}(v)$, by Definition 3, at time $t_v$, $v$ generates a conflict with $T'$. The other two cases, where $R(v,\ell) \wedge W(w,\ell)$ or $W(v,\ell) \wedge W(w,\ell)$, are analogous.

□

***Prefix Race-Freedom Implies Serializability by Modules***

Finally, we show that a trace $(\mathcal{C},\Phi)$ which is prefix race-free is also serializable by modules.

**THEOREM 8.** *Any trace $(\mathcal{C},\Phi)$ which is prefix race-free is also serializable by modules.*

PROOF.

First, by Definition 11 and Lemma 4, it is easy to see that a prefix-race free sort $\mathcal{S}$ of a trace $(\mathcal{C},\Phi)$ is also prefix-race free of the sort $(\texttt{mTree}(\mathcal{C},A),\Phi)$ for any Xmodule $A$. Now we shall argue that for any Xmodule $A$, we can transform $\mathcal{S}$ into $\mathcal{S}_A$ such that all transactions in $\texttt{xactions}(A)$ appear contiguous in $\mathcal{S}_A$.

Consider a prefix-race free sort $\mathcal{S}$ of $(\texttt{mTree}(\mathcal{C},A),\Phi)$ which has $k$ nodes $v$ which violate the second condition of Definition 12. We show how to construct a new order $\mathcal{S}'$ which is still a prefix race-free sort of $(\texttt{mTree}(\mathcal{C},A),\Phi)$, but which has only $k-1$ violations.

We reduce the number of violations according to the following procedure:

1. Of all transactions $T \in \texttt{modXactions}(A)$ such that there exists an operation $v$ such that $\texttt{xbegin}(T) \leq_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{xend}(T)$ and $v \notin V(T)$, choose the $T = T^*$ which has the latest $\texttt{xend}(T)$ in the order $\mathcal{S}$.
2. In $T^*$, pick the first $v \notin V(T^*)$ which causes a violation.
3. Create a new sort $\mathcal{S}'$ by moving $v$ to be immediately before $\texttt{xbegin}(T^*)$.

In order to argue that $\mathcal{S}'$ is still a prefix race-free sort of $(\texttt{mTree}(\mathcal{C},A),\Phi)$, we need to show that moving $v$ does not generate any new prefix races, and does not create a sort $\mathcal{S}'$ which is no longer sequentially consistent with respect to $\Phi$ (i.e., that $\Phi$ is still the transactional last writer according to $\mathcal{S}'$). There are three cases: $v$ can be a memory operation, an $\texttt{xbegin}(T')$, or an $\texttt{xend}(T')$.

1. Suppose $v$ is a memory operation which accesses location $\ell$. For all operations $w$ such that $\texttt{xbegin}(T) <_{\mathcal{S}} w <_{\mathcal{S}} v$, we argue that $w$ can not access the same location $\ell$ unless both $w$ and $v$ read from $\ell$. Since we chose $v$ to be the first memory operation such that $\texttt{xbegin}(T) <_{\mathcal{S}} v <_{\mathcal{S}} \texttt{xend}(T)$ such that $v \notin V(T)$, we know $w \in V(T)$. We know by construction of $\texttt{mTree}(\mathcal{C},A)$, that $w \in \texttt{cContent}(T)$ (if $w \in \texttt{oContent}(T)$ or $w \in \texttt{aContent}(T)$, then steps 1 or 2, respectively, in Definition 11 will turn $w$ into a nop). Therefore, by Definition 13, unless $w$ and $v$ both read from $\ell$, $v$ has a prefix race with $T$, contradicting the fact that $\mathcal{S}$ is a prefix race-free sort of the trace. Thus, moving $v$ to be before $\texttt{xbegin}(T)$ can not generate any new prefix races or change the transactional last writer for any memory operation, and $\mathcal{S}'$ is still a prefix race-free sort of the trace.
2. Next, suppose $v = \texttt{xbegin}(T')$. Moving $\texttt{xbegin}(T')$ can not generate any new prefix races with $T'$, because the only memory operations $u$ which satisfy $\texttt{xbegin}(T) <_{\mathcal{S}} u <_{\mathcal{S}} \texttt{xbegin}(T')$ satisfy $u \notin \texttt{cContent}(T')$. Also, moving $\texttt{xbegin}(T')$ does not change the transactional last writer for any node $v$ because the move preserves the relative order of all memory operations. Therefore, $\mathcal{S}'$ is still a prefix race-free sort.
3. Finally, suppose $v = \texttt{xend}(T')$. By moving $\texttt{xend}(T')$ to be before $\texttt{xbegin}(T)$, we can only lose prefix races with $T'$ that already existed in $\mathcal{S}$ because we are moving nodes out of the interval $[\texttt{xbegin}(T'),\texttt{xend}(T')]$. Also, as with $\texttt{xbegin}(T')$, moving $\texttt{xend}(T')$ does not change any transaction last writers. Therefore, $\mathcal{S}'$ is still a prefix race-free sort of the trace.

Since we can eliminate violations of the second condition of Definition 12 one at a time, we can construct a sort $\mathcal{S}_A$ which satisfies serializability by modules by eliminating all violations. □

Jim: This proof is probably st

Finally, we can prove the *OAT* model guarantees serializability by modules by putting the previous results together.

**THEOREM 9.** *Any trace $(\mathcal{C},\Phi)$ generated by the OAT model is serializable by modules.*

PROOF. By Theorem 7, the *OAT* model generates only trace $(\mathcal{C},\Phi)$ which are prefix race-free. By Theorem 5.3, any trace $(\mathcal{C},\Phi)$ which is prefix race-free is serializable by modules. □

### 5.4 Abstract Serializability

By Theorem 9, the *OAT* model guarantees serializability by modules. We now relate this definition to the notion of *abstract serializability* used in multilevel database systems [11]. As we mentioned in Section 1, ownership-based commit mechanism forms a part of a methodology which includes abstract locks and compensating actions. In this section we argue that *OAT* model provides enough flexibility to accommodate abstract locks and compensating actions. In addition, if a program is "properly locked and compensated," then serializability by modules guarantees abstract serializability.

The definition of abstract serializability in [11] assumes that the program is divided into levels and a transaction at level *i* can only call a transaction at level $i-1$. In addition, transactions at a particular level have predefined commutativity rules, i.e., some transactions of the same Xmodule can commute with each other and some can not. These commutativity rules might be specified using abstract locks [9]: if two transactions grab the same abstract lock in a conflicting manner, then they cannot be reordered. Using the application in Section 1 for instance, transactions calling `insert` and `remove` on the `BST` using the same key do not commute and should grab the same write lock.

The transactions at level 0 are naturally serializable. Given this schedule $\mathcal{Z}_0$ of level-0 transactions, the schedule is said to be serializable at level 1 if all transactions in $\mathcal{S}_0$ can be reordered, obeying all commutativity rules, so that we can construct a serializable order for level-1 transactions. This order of level-1 transactions can be called $\mathcal{Z}_1$. Similarly, for level-*i* transactions, reorder $\mathcal{Z}_{i-1}$ of level-$i-1$ transactions, obeying all commutativity rules, so that we get a serializable order for level-*i* transactions. Continuing in this way up to the top-level transactions, the original schedule is said to be abstractly serializable if it is serializable for all levels.

This definition holds for our model in the special case when the module tree is a chain (i.e., each non-leaf module has exactly one child). A transaction $T$ is at level $i$ if $\texttt{level}(\texttt{xMod}(T)) = i$. Although abstract locks are not explicitly modeled in the *OAT* model, simple read/write locks can be modeled as reads and writes to memory locations.[4] We can think transactions acquiring the same abstract lock as them writing to a common memory location $\ell$. Locks associated with an Xmodule $A$ are owned by $\texttt{modParent}(A)$. A module $A$ is said to be ***properly locked*** if the following is true for all transactions $T_1, T_2$ with $\texttt{xMod}(T_1) = \texttt{xMod}(T_2) = A$: if $T_1$ and $T_2$ do not commute, then they access some $\ell \in \texttt{modMemory}(\texttt{modParent}(A))$ in a conflicting manner. In the special case when the module tree is a chain, one can show that if all modules are properly locked, then serializability by modules implies abstract serializability.

In the general case, however, a transaction at level *i* can call transactions at many levels, not just $i-1$. By Rule 2 of Definition 1, however, we know that transactions at level *i* can only call transactions at a lower levels. Thus, we change our definition slightly. Instead of reordering just $\mathcal{S}_{i-1}$ while serializing transactions at level-*i*, we have to potentially reorder $\mathcal{S}_x$ for all *x* where transactions at level *i* can call transactions at level *x*. Even in this case, the module tree properties guarantee that if every module is properly locked (by the same definition as above), serializability by modules guarantees abstract serializability.

cks work the same way?

The methodology of open-nesting in TM often requires the notion of compensating actions or inverse actions. For instance, the inverse of `BST.insert` is `BST.remove` with the same key. When a transaction $T$ aborts, all the changes made by its subtransactions must be inverted. Again, although *OAT* model does not explicitly model compensating actions, it allows an aborting transaction with status `PENDING_ABORT`

[4] More complicated locks can be modeled by generalizing the definition of conflict.

to perform an arbitrary but finite number of operations before changing the status to ABORTED. Therefore, an aborting transaction can compensate for all its aborted subtransactions. *OAT* model does not place any restrictions on the order of execution of compensating actions.

## 6. DEADLOCK FREENESS

In this section, we argue that the *OAT* model we described in Section 4 can never enter a "semantic deadlock" if we impose suitable restrictions on the memory that a transaction's abort actions can access. In particular, an abort action for a transaction $T$ from $\texttt{xMod}(T)$ can read (write) from a memory location $\ell$ belonging to $\texttt{modAnces}(\texttt{xMod}(T))$ if $\ell$ is already in $\texttt{R}(T)$ ($\texttt{W}(T)$).[5] Under these conditions, we show that the *OAT* model can always "finish" reasonable computations.

Intuitively, an ordinary TM without open nesting and with eager conflict detection never enters a semantic deadlock because it is always possible to finish aborting a transaction $T$ without generating additional conflicts. Thus, a scheduler in the TM runtime could abort all transactions, and then complete the computation by running the remaining transactions serially. Using the *OAT* model, however, a TM system can enter a semantic deadlock because it can enter a state in which it is impossible to finish aborting two parallel transactions $T_1$ and $T_2$ which both have status PENDING_ABORT. If $T_1$'s abort action generates a memory operation $u$ which conflicts with $T_2$, then $u$ will wait for $T_2$ to finish aborting and change its status to ABORTED. Similarly, $T_2$'s abort action can generate an operation $v$ which conflicts with $T_1$ and waits for $T_1$ to finish aborting. Since $T_1$ and $T_2$ are both waiting on each other, neither transaction will ever finish aborting.

### *Defining Semantic Deadlock*

Intuitively, we want to say that the *OAT* model exhibits a semantic deadlock if it causes the TM system state machine to enter a state in which it is impossible to "finish" a computation because of transaction conflicts. A computation might not finish for other reasons, such as an infinite loop or livelock. This section defines semantic deadlock precisely and distinguishes it from these other reasons for noncompletion.

Recall that our abstract model has two entities: the program, and a generic operational model $N$ representing the runtime system. At any time $t$, given a ready node $X \in \texttt{ready}(\mathcal{C})$, the program chooses an instruction and has $X$ issue the instruction. If the program issues an infinite number of instructions, then $N$ can not complete the program no matter what it does. To eliminate programs which have infinite loops, we only consider ***bounded programs***.

**DEFINITION 15.** *We say that a program is **bounded** for an operational model $N$ if any computation tree that $N$ generates for that program is of a finite depth, and there exists a finite number $K$ such that at any time $t$, every node $B \in \texttt{nodes}(t, \mathcal{C})$ has at most $K$ children with status PENDING, COMMITTED or PENDING_ABORT.*

Notice that this definition does not disallow infinite number of aborted transactions, since even a computation without an infinite loop may have to re-execute a transaction an infinite number of times if the $N$ keeps aborting the transaction. However, there is no reason to have in infinite number of pending or committed transactions unless the computation is infinite. [6]

Another reason a program might run forever is if an operational model makes bad scheduling decisions. An operational model $N$ makes two types of nondeterministic choices. First, at any time $t$, $N$ nondeterministically chooses which ready node $X \in \texttt{ready}(\mathcal{C})$ executes an instruction. This choice models nondeterminism in the program due to interleaving of the parallel executions. Second, while performing a memory operation $u$ which generates a conflict with transaction $T$, $N$ nondeterministically chooses to abort either $\texttt{xparent}(u)$ or $T$. This nondeterministic choice models the contention manager of the TM runtime. A program may run

[5] Roughly, this translates into restrictions on the compensating actions as follows: A compensating action for transaction $T'$ can not access any new memory belonging to higher level modules.

[6] We assume that if a transaction aborts, it is not retried until it finishes aborting. That is, a transaction is retried only after its status changes to ABORTED.

forever due to ***livelock*** if $N$ repeatedly makes "bad" choices. For example, two transactions may continually abort each other due to retries, causing the program to run forever.

An intelligent scheduler, however, might be able to avoid a livelock. Therefore, we use a notion of ***schedule*** to distinguish a livelocks from a semantic deadlock.

**DEFINITION 16.** *A **schedule** $\Gamma$ on some time interval $[t_0, t_1]$ is the sequence of nondeterministic choices made by an operational model in the interval.*

Intuitively, an operational model deadlocks if it allows a bounded computation to reach a state where no schedule can complete the computation after this point. Notice that this definition excludes livelocks since livelocks can be solved by good subsequent scheduling decisions, while deadlocks can not be.

**DEFINITION 17.** *Consider an $N$ executing a bounded computation. We say that $N$ does not exhibit a **semantic deadlock** if for all finite sequences of $t_0$ instructions that $N$ can issue that generates some intermediate computation tree $\mathcal{C}_0$, there exists a finite schedule $\Gamma$ on $[t_0, t_1]$ such that $N$ brings the computation tree to a rest state $\mathcal{C}_1$, i.e., $\texttt{ready}(\mathcal{C}_1) = \{\texttt{root}(\mathcal{C}_1)\}$.*

This definition is sufficient, since once the computation tree is at the rest state, and only the root node is ready, $N$ can execute each transaction serially and complete the computation.

***Restrictions to Avoid Semantic Deadlock***

The general *OAT* model described in Section 4 exhibits semantic deadlock because it is possible to enter a state where two parallel aborting transactions $T_1$ and $T_2$ keep each other from completing their aborts. But for a restricted set of programs, where a `PENDING_ABORT` transaction never accesses new memory belonging to high-level modules, we can show the *OAT* model is free of semantic deadlock.

More formally, for all transactions $T$, we restrict the memory footprint of $\texttt{abortactions}(T)$.

**DEFINITION 18.** *An execution (represented by a computation tree $\mathcal{C}$) has **abort actions with limited footprint** if the following condition is true for all transactions $T \in \texttt{aborted}(\mathcal{C})$. At time $t$, if a memory operation $v \in \texttt{abortactions}(T)$ accesses location $\ell$ and $\texttt{owner}(\ell) \in \texttt{modAnces}(\texttt{xMod}(T))$, then (1) if $R(v,\ell)$ then $\ell \in \texttt{R}(T)$, and (2) if $W(v,\ell)$ then $\ell \in \texttt{W}(T)$.*

Intuitively, Definition 18 requires that once a transaction $T$'s status becomes `PENDING_ABORT`, any memory operation $v$ which $T$ or a nested transaction inside $T$ performs to finish aborting $T$ can not read from (write to) any location $\ell$ which is owned by any Xmodules which are ancestors of $\texttt{xMod}(T)$, unless $\ell$ is already in the in the readset (writeset) of $T$.

First, we show that the properties of Xmodules from Theorem 2 in combination with the ownership-aware commit mechanism imply that transaction readsets and writesets exhibit nice properties. In particular, we have Corollary 10, which states that a location $\ell$ can appear in the readset of a transaction $T$ only if $T$'s Xmodule is a descendant of $\texttt{owner}(\ell)$ in the module tree $\mathcal{D}$.

**COROLLARY 10.** *For any transaction $T$ if $\ell \in \texttt{R}(T)$, then $\texttt{xMod}(T) \in \texttt{modDesc}(\texttt{owner}(\ell))$.*

PROOF. Follows from Definition 1 and Theorem 2, and induction on how a location $\ell$ can propagate into readsets and writsets using the ownership-aware commit mechanism. □

If all abort actions have a limited footprint, we can show that operations of an abort action of an Xmodule *A* can only generate conflicts with a "higher-level" Xmodule *B*.

**LEMMA 11.** *Suppose the OAT model generates an execution where abort actions have limited footprint. For any transaction $T$, consider a potential memory operation $v \in \texttt{abortactions}(T)$. If $v$ conflicts with transaction $T'$, then $\texttt{level}(\texttt{xMod}(T')) < \texttt{level}(\texttt{xMod}(T))$.*

PROOF. Suppose $v \in \texttt{abortactions}(T)$ accesses a memory location $\ell$ with $\texttt{owner}(\ell) = A$. Since $\texttt{abortactions}(T) \subseteq \texttt{memOps}(T)$, by the properties of Xmodules given in Definition 1, we know that either

$A \in \texttt{modAnces}(\texttt{xMod}(T))$, or $\texttt{level}(A) < \texttt{level}(\texttt{xMod}(T))$. If $A \in \texttt{modAnces}(\texttt{xMod}(T))$, then by Definition 18, $T$ already had $\ell$ in its read or write set. Therefore, using Definition 3, $v$ can not generate a conflict with $T'$ because then $T$ would already have had a conflict with $T'$ before $v$ occurred, contradicting the eager conflict detection of the *OAT* model.

Thus, we have $\texttt{level}(A) < \texttt{level}(\texttt{xMod}(T))$. If $v$ conflicts with some other transaction $T'$, then $T'$ has $\ell$ in its read or write set. Therefore, from Corollary 10, $\texttt{xMod}(T') \in \texttt{modDesc}(A)$. Thus, we have $\texttt{level}(\texttt{xMod}(T')) < \texttt{level}(A) < \texttt{level}(\texttt{xMod}(T))$. □

**THEOREM 12.** *In the case where aborted actions have limited footprint, the OAT model is free from semantic deadlock.*

PROOF. Let $\mathcal{C}_0$ be the computation tree after any finite sequence of $t_0$ instructions. We describe a schedule $\Gamma$ which finishes aborting all transactions in the computation by executing abort actions and transactions serially.

Without loss of generality, assume that at time $t_0$, all active transactions $T$ have $\texttt{status}[T] = \texttt{PENDING\_ABORT}$. Otherwise, the first phase of the schedule $\Gamma$ is to make this status change for all active transactions $T$. [7]

For a module tree $\mathcal{D}$ with $k$ Xmodules, the schedule $\Gamma$ has $k$ phases, $0, 1, \ldots k-1$, one for each Xmodule in $\mathcal{D}$, starting at the lowest level Xmodule. The invariant we maintain is that immediately before phase $i$, we bring the computation tree into a state $\mathcal{C}^{(i)}$ which has no active transaction instances $T$ with $\texttt{level}(\texttt{xMod}(T)) < i$, i.e., no instances $T$ from Xmodules at level lower than $i$.

In the proof, let $\beta_i$ denote the subset of all active transaction instances $T$ that are generated by Xmodule at level $i$. In other words,

$$\beta_i(t) = \{T \in \texttt{xactions}(\mathcal{C}) \cap \texttt{activeN}(t, \mathcal{C}) \ : \ \texttt{level}(\texttt{xMod}(T)) = i\}.$$

By induction, we show that if after phase $i$, for all $j$ where $j < i$, $\beta_j(t) = \emptyset$, then after phase $i$ schedule $\Gamma$ makes $\beta_i(t) = \emptyset$, after some finite number of steps.

In the base case, consider the Xmodule $A$ at the lowest level ($\texttt{level}(A) = 0$). We know, from Definition 1 that $T \in \beta_0$ has no nested subtransactions, since a transaction from module $A$ can only call transactions from a module at a lower level.

First, we claim that aborting any transaction $T \in \beta_0$ never causes any conflicts. By Lemma 11, we know that if $v \in \texttt{abortactions}(T)$ causes a conflict with transaction $T'$, then $\texttt{level}(\texttt{xMod}(T')) < \texttt{level}(\texttt{xMod}(T))$. But $\texttt{xMod}(T)$ has level 0. Therefore $T$ completes aborting eventually without generating any new conflicts. By Definition 15, there are a finite number of these transactions $T$ in $\beta_0$, and each of these transactions can generate a finite number of abort actions. Thus, in the *OAT* model, $\Gamma$ can finally issue an $\texttt{xabort}$ for all $T \in \beta_0$ and in some finite number of time steps, phase 1 of $\Gamma$ can make $\beta_0 = \emptyset$.

In the inductive step, assume before phase $i$ at time $t$, $\beta_j(t) = \emptyset$ for all $j < i$. Pick any transaction $T \in \beta_i(t)$. By the inductive hypothesis, we know that there are no active transactions $T'$ with $\texttt{level}(\texttt{xMod}(T')) < \texttt{level}(\texttt{xMod}(T))$. Therefore by Lemma 11, we can conclude that $\Gamma$ can finish aborting $T$ in a finite amount of time without generating any new conflicts. Therefore $\Gamma$ can abort all such $T$ serially in a finite number of steps.

After phase $k-1$ of the scheduling algorithm $\Gamma$, we have $\beta_i = \emptyset$ for all $i < k$. Thus, we only left with the root transaction $\texttt{root}(\mathcal{C})$ from the Xmodule $\texttt{world}$ , completing the proof.

□

[7] A slightly less wasteful serial scheduler in this case can be lazy and issue a $\texttt{sigabort}$ to $T$ if and when the first conflict to $T$ is discovered; the rest of the proof still works assuming that $\texttt{PENDING\_ABORT}$ transactions of the same Xmodule are all scheduled and completed before attempting to finish $\texttt{PENDING}$ transactions.

***Restrictions on compensating actions***

If transactions $Y_1, Y_2 ...$ are nested inside transaction $X$ and $X$ aborts, typically abort actions of $X$ simply consists of compensating actions for $Y_1, Y_2 ...$ Therefore, restrictions on abort actions translate in a straightforward manner to restrictions on compensating actions: A compensating action for a transaction $Y_1$ should not access any memory owned by $\texttt{xMod}(X)$ or its ancestors unless the memory location is already in $X$'s read/write set. Assuming locks are modeled as accesses to memory locations, the same restriction applies, meaning, a compensating action can not acquire new locks that were not already acquired by the transaction it is compensating for.

## 7. CONCLUSIONS

In this paper, we have bridged the gap between the intent and the execution of open-nested transactions. Open-nested transactions are meant to allow the TM to ignore low-level memory conflicts while doing conflict detection on high-level transactions. We have described a framework that incorporates the notions of high-level and low-level in the specification of the program, thus allowing a transactional memory system to make the right decisions about which memory conflicts should be ignored.

We have described a framework that incorporates the notions of Xmodules and ownership into a TM system. We propose precise restrictions that must be imposed on the interactions between Xmodules. In addition, we introduce the ownership-aware commit mechanism which commits memory selectively based on which Xmodule owns that piece of memory. If a program follows all the restrictions we detailed and the TM system uses the ownership-aware commit mechanism, we prove that the system will guarantee serializability by modules. Finally, it might be difficult for the programmer to make sure that they have followed all the restrictions outlined. Therefore, we propose a type system that allows the compiler to check that the programmer has obeyed all the restrictions needed by the ownership-aware transactional memory system.

## REFERENCES


[1] K. Agrawal, C. E. Leiserson, and J. Sukha. Memory models for open-nested transactions. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, October 2006. In conjunction ASPLOS.

[2] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, Louisiana, Jan. 2003.

[3] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 56–67, New York, NY, USA, 2007. ACM Press.

[4] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 289–300, 2003.

[5] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2006.

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

[7] J. E. B. Moss. Open nested transactions : Semantics and support. In *Proceedings of the Workshop on Memory Performance Issues (WMPI)*, Austin, Texas, Feb 2006.

[8] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and architecture sketches. In *Science of Computer Programming*, volume 63, pages 186–201. Elsevier, Dec 2006.

[9] Y. Ni, V. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, Mar. 2007.

[10] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[11] G. Weikum. A theoretical foundation of multi-level concurrency control. In *Proceedings of the ACM SIGACT-SIGMOD symposium on Principles of database systems (PODS)*, pages 31–43, New York, NY, USA, 1986. ACM Press.

# A. THE *OAT* MODEL AND SEQUENTIAL CONSISTENCY

This appendix contains the details of the proof of Theorem 5, that if the *OAT* model generates a trace $(\mathcal{C}, \Phi)$ and a topological sort order $\mathcal{S}$, that $\mathcal{S}$ satisfies Definition 9, i.e., $\mathcal{S}$ is sequentially consistent with respect to $\Phi$.

In this appendix, we first define some useful notation for the proof. Next, we prove that the *OAT* model preserves several invariants about memory operations, readset, and writesets. Finally, we use these invariants to prove Theorem 5.

## A.1 Notation

We define some notation that is useful later for stating operational invariants of the *OAT* model.

For any subset $S$ of nodes in the computation tree $\mathcal{C}$, i.e., $S \subseteq \texttt{nodes}(\mathcal{C})$, define

- $\texttt{low}(S) = \{X \in S : \texttt{pDesc}(X) \cap S = \emptyset\}$.
- $\texttt{high}(S) = \{X \in S : \texttt{pAnces}(X) \cap S = \emptyset\}$.

Intuitively, $\texttt{low}(S)$ represents the nodes in $S$ closest to the leaves of the tree. Similarly, $\texttt{high}(S)$ represents the nodes in $S$ closest to the root of the tree. In cases where the set $S$ is guaranteed to fall along one root-to-leaf path in the tree, we define $\texttt{lowest}(S)$ as the only element $X \in \texttt{low}(S)$. Similarly, we define $\texttt{highest}(S)$ as the only element in $\texttt{high}(S)$.

We also define two time-dependent sets of transactions.

- The ***reader set*** $\texttt{readers}(t, \ell) = \{T \in \texttt{activeX}(t, \mathcal{C}) : \ell \in \texttt{R}(t, T)\}$.
- The ***writer set***, $\texttt{writers}(t, \ell) = \{T \in \texttt{activeX}(t, \mathcal{C}) : \ell \in \texttt{W}(t, T)\}$.

Said differently, $\texttt{readers}(t, \ell)$ is the set of active transactions at time $t$ which have location $\ell$ in their readset. Similarly, $\texttt{writers}(t, \ell)$ is the set of active transactions at time $t$ with $\ell \in \texttt{W}(T)$.

Next, we generalize the content sets from Definition 6 and define a set of dynamic content sets.

**DEFINITION 19.** *At any time $t$, for any transaction $T \in \texttt{xactions}(t, \mathcal{C})$ and a memory operation $u \in \texttt{memOps}(t, \mathcal{C})$, define the sets $\texttt{cContent}(t, T)$, $\texttt{oContent}(t, T)$, $\texttt{aContent}(t, T)$, and $\texttt{vContent}(t, T)$ according the $\texttt{ContentType}(t, u, T)$ procedure:*

```
   ContentType(t,u,T)        ▷ For any u ∈ memOps(t,T)
1  X ← xparent(u)
2  while (X ≠ T)
3     if X ∈ activeX(t,C),        return u ∈ vContent(t,T)
4     if X ∈ aborted(t,C),        return u ∈ aContent(t,T)
5     if (X = committer(u))       return u ∈ oContent(t,T)
6     X ← xparent(X)
7  return u ∈ cContent(t,T)
```

The difference between Definition 19 and the previous statement in Definition 6 is that for dynamic content sets, if we encounter a `PENDING` or `PENDING_ABORT` transaction when walking up the tree from a memory operation $u$ to a transaction $T$, we place $u$ in the *active content* of $T$, i.e., $u \in \texttt{vContent}(t, T)$. If a transaction $T$ completes at time $t^*$, it is not hard to see that the dynamic classification $\texttt{ContentType}(t, u, T)$ gives the same answer as the static classification $\texttt{ContentType}(u, T)$ for all times $t \geq t^*$.

Finally, we define subsets of the dynamic content sets which write to a particular memory location.

## A.2 *OAT* Model Invariants

Because the *OAT* model performs eager conflict detection according to Definition 3, it is not hard to prove the following invariant about the readers and writers to a particular memory location $\ell$.

**THEOREM 13.** *At all times $t$, the OAT maintains the following invariants on the sets* $\texttt{readers}(\ell)$ *and* $\texttt{writers}(\ell)$*:*

1. *For all* $\ell \in \mathcal{L}$, $|\texttt{low}(\texttt{writers}(t,\ell))| = 1$, *i.e.,* $\texttt{lowest}(\texttt{writers}(t,\ell))$ *exists.*
2. *For any* $T \in \texttt{readers}(t,\ell)$, *either* $\texttt{lowest}(\texttt{writers}(t,\ell)) \in \texttt{desc}(T)$ *or* $T \in \texttt{desc}(\texttt{lowest}(\texttt{writers}(t,\ell)))$.

PROOF. The proof is by induction on the instructions that the *OAT* model issues.

In the base case, for all locations $\ell \in \mathcal{L}$, we begin with $\texttt{readers}(0,\ell) = \texttt{writers}(0,\ell) = \{\texttt{root}(\mathcal{C})\}$, and no other nodes in the computation tree $\mathcal{C}$ except $\texttt{root}(\mathcal{C})$. Thus, Invariants 1 and 2 are satisfied.

In the inductive step, suppose at time $t-1$, Invariants 1 and 2 are satisfied. A `read` or `write` instruction at time $t$ can not break the invariants without causing a conflict according to Definition 3. Therefore, successful `read` and `write` operations preserve the invariant. An unsuccessful `read` or `write` operation can only trigger the `sigabort` of transactions, which does not affect either invariant.

An `xend` instruction that commits a transaction $T$ can only add the transaction $\texttt{xparent}(T)$ to $\texttt{readers}(\ell)$ or $\texttt{writers}(\ell)$. Since $\texttt{xparent}(T)$ is an ancestor of $T$, it can not break either of the two invariants.

The remaining instructions preserve Invariants 1 and 2 trivially. A `fork` or `join` instruction at time $t$ preserves the invariants because they do not change the set active transactions or any transaction readsets or writesets. An `xbegin` preserves the invariants because it creates new transactions $T$ with empty readsets and writesets. The `xabort` instruction preserves the invariants because it can only remove transactions from $\texttt{readers}(t,\ell)$ or $\texttt{W}(t,\ell)$. □

Jim: This proof could be bette

The following invariant shows that, informally, the readsets of transactions act as caches for pairs $(\ell,u)$ stored in writesets.

**LEMMA 14.** *At any time $t$, for any* $T \in \texttt{readers}(t,\ell)$, *suppose* $(\ell,u) \in \texttt{R}(t,T)$. *Let* $T' = \texttt{lowest}(\texttt{xAnces}(T) \cap \texttt{writers}(t,\ell))$. *Then* $(\ell,u) \in \texttt{W}(t,T')$.

PROOF. The proof is by induction on the instructions issued by the *OAT* model. In the base case, we know for all memory locations $\ell \in \mathcal{L}$,, we start with $\texttt{readers}(0,\ell) = \texttt{writers}(0,\ell) = \{\texttt{root}(\mathcal{C})\}$ and $\texttt{R}(\texttt{root}(\mathcal{C})) = \texttt{W}(\texttt{root}(\mathcal{C}))$. Since $T' = T = \texttt{root}(\mathcal{C})$, Lemma 14 is satisfied in the base case.

For the inductive step, assume the lemma is satisfied at time $t-1$. We show after any *S*-node $X$ issues an instruction at time $t$, the lemma is still satisfied.

For any $T \in \texttt{xactions}(t-1,\mathcal{C})$, after a `fork`, `join`, or `xbegin` instruction in step $t$, we have $\texttt{R}(t,T) = \texttt{R}(t-1,T)$ and $\texttt{W}(t,T) = \texttt{W}(t-1,T)$. Thus, the lemma is satisfied after these instructions. An `xbegin` which creates a new transaction $X$ at time step $t$ starts with $\texttt{R}(t,X) = \texttt{W}(t,X) = \emptyset$; thus, the lemma is satisfied.

Next, consider an `xabort` issued by $X \in \texttt{xactions}(t-1,\mathcal{C})$. Suppose, before the `xabort` of $X$ there exists a transaction $T \in \texttt{readers}(t-1,\ell)$ with $(\ell,u) \in \texttt{R}(t-1,T)$. Let $T' = \texttt{lowest}(\texttt{xAnces}(T) \cap \texttt{writers}(t-1,\ell))$. Then before the `xabort`, $(\ell,u) \in \texttt{W}(t-1,T')$. Assume for contradiction after the `xabort` of $X$, that there exists some transaction $T \in \texttt{xactions}(t,\mathcal{C})$ such that the invariant no longer holds for $T$, i.e., we no longer have $(\ell,u) \in \texttt{W}(t,T')$. Since an `xabort` does not change the contents of any transaction's writeset, but removes $X$ from $\texttt{writers}(\ell)$, the only way to violate the invariant is if $X = T'$. Consider two cases: either $X = T' = T$, or $X = T' \neq T$. In the first case, we can not violate the invariant for $T$ because $T$ is aborted and removed from $\texttt{readers}(\ell)$. In the second case, we must have $T \in \texttt{pDesc}(X)$. But then, before the `xabort`, we have $T \in \texttt{pDesc}(X) \cap \texttt{activeN}(t-1)\mathcal{C}$ and $X \in \texttt{ready}(t-1)\mathcal{C}$, contradicting Property 2, that the ready nodes are the leaves of tree of active nodes. Thus, the `xabort` must preserve the invariant.

Finally, suppose at time $t$, a ready node $X$ issues an `xend`. Consider two cases:

1. $X \neq \texttt{owner}(\ell)$. The only transaction $Y$ for which we could have $\texttt{R}(t,Y) \neq \texttt{R}(t-1,Y)$ or $\texttt{W}(t,Y) \neq \texttt{W}(t-1,Y)$ is $Y = \texttt{xparent}(X)$. Thus, after the `xend`, for all $T \in \texttt{readers}(t,\ell)$ with $T \neq Y$, since the readset or writeset of $T$ or any transaction in $\texttt{xAnces}(T)$ remains the same, the invariant is still preserved for $T$.
2. Suppose $X = \texttt{owner}(\ell)$. Then, the only transaction whose readset or writeset can change is $Y = \texttt{root}(\mathcal{C})$. But the only way to break the invariant is if $X$ commits a pair $(\ell,v)$ to $\texttt{root}(\mathcal{C})$, which corrupts the version

$(\ell,u) \in \mathtt{R}(t-1,T)$, for some parallel transaction $T$. But then, we would violate Theorem 13, and should have had a conflict earlier.

Since all possible choices for action $k+1$ preserve the invariant, the lemma holds by induction. □

Theorem 15 characterizes when a transaction should have a location in its writeset.

**THEOREM 15.** *At any time $t$, consider any transaction $T \in \mathtt{activeX}(t,\mathcal{C})$ and any memory location $\ell$ such that $\mathtt{level}(\mathtt{owner}(\ell)) \geq \mathtt{xMod}(T)$. Let $S_\ell(t) = \{u \in \mathtt{memOps}(t,\mathcal{C}) \ : \ W(u,\ell)\}$. Exactly one of the following cases holds:*

1. *$T = \mathtt{root}(\mathcal{C})$, $(\ell,\bot) \in \mathtt{W}(t,T)$, and two conditions are satisfied:*
   *(a) $\mathtt{cContent}(t,T) \cap S_\ell = \emptyset$.*
   *(b) For all $v \in S_\ell(t)$, we have $v \in \mathtt{aContent}(t,T) \cup \mathtt{vContent}(t,T)$.*
2. *There exists an $(\ell,u) \in \mathtt{W}(t,T)$ which happens at time $t_u$, and two conditions are satisfied:*
   *(a) $u \in \mathtt{cContent}(t,T) \cap S_\ell(t)$*
   *(b) For any operation $v \in (S_\ell(t) - \{u\})$ which happens at time $t_v$, where $t_u < t_v \leq t$, we have $v \in \mathtt{aContent}(t,T) \cup \mathtt{vContent}(t,T)$.*
3. *We have $\ell \notin \mathtt{W}(t,T)$, and $\mathtt{cContent}(t,T) \cap S_\ell(t) = \emptyset$.*

PROOF.

This theorem can be proved by a straighforward, albeit tedious, induction on time.

Note that because we assume $\mathtt{level}(\mathtt{owner}(\ell)) \geq \mathtt{xMod}(T)$, $S_\ell(t) \cap \mathtt{memOps}(t,\mathcal{C}) \cap \mathtt{oContent}(t,T) = \emptyset$, i.e., the theorem is only concerned with memory locations $\ell$ which belong to $T$'s open content. Because of the properties of ownership and Xmodules, any location $\ell$ with $\mathtt{level}(\mathtt{owner}(\ell)) < \mathtt{xMod}(T)$ can never propagate into $T$'s writeset anyway. □

The intuition for Theorem 15 is that if at time $t_u$, a pair $(\ell,u)$ appears in the writeset of a transaction $T$, then all other $v$ which write to $\ell$ which happen after time $t_u$ are in $T$'s subtree, and $v \in \mathtt{aContent}(t,T) \cup \mathtt{vContent}(t,T)$ (i.e., $v$ is aborted or still pending with respect to $T$).

### A.3 Proof of Sequential Consistency

Finally, we can use the invariants from Lemma 14 and Theorem 15 to prove Theorem 5.

PROOF. [Theorem 5]

The first condition and second conditions are true by construction, since the *OAT* model can only set $\Phi(v) = u$ if $u <_\mathcal{S} v$, $W(u,\ell)$ and $R(v,\ell) \wedge W(v,\ell)$.

To check the third and fourth conditions, we require some setup. Suppose at time $t = \mathcal{S}(v)$, the *OAT* model sets $\Phi(v) = u$. Let $A = \mathtt{lowest}(\mathtt{readers}(t,\ell) \cap \mathtt{ances}(v))$. Because the *OAT* model sets $\Phi(v) = u$, we must have $(\ell,u) \in \mathtt{R}(t,A)$. Let $T = \mathtt{lowest}(\mathtt{xAnces}(A) \cap \mathtt{writers}(t,\ell))$. By Lemma 14, we know $(\ell,u) \in \mathtt{W}(t,T)$. By Theorem 15, since $(\ell,u) \in \mathtt{W}(t,T)$, we know $u \in \mathtt{cContent}(t,T)$. Let $X = \mathtt{xLCA}uv$. We must have $T \in \mathtt{ances}(X)$; otherwise, we could not have $\{u,v\} \subseteq \mathtt{memOps}(t,T)$.

Since $u \in \mathtt{cContent}(t,T)$, we know $u \in \mathtt{cContent}(t,X) \cup \mathtt{oContent}(t,X)$. Therefore, we have $\neg(uHv)$, satisfying the third condition.

To check the fourth condition, assume for contradiction that there exists a $w$ such that $W(w,\ell)$, and $u <_\mathcal{S} w <_\mathcal{S} v$. Let $t_v$ be the time that $v$ happens. Then, since $\Phi(v) = u$, we know $u \in \mathtt{W}(t_v,T)$. Therefore, by Theorem 15 we know $w \in \mathtt{memOps}(t_v,T)$, $w \in \mathtt{aContent}(t_v,T) \cup \mathtt{vContent}(t_v,T)$.

Let $Y = \mathtt{xLCA}wv$. Since $w \in \mathtt{memOps}(t_v,T)$, we know $T \in \mathtt{ances}(Y)$. Consider the two cases for $w$:

1. Suppose $w \in \mathtt{aContent}(t_v,T)$. Since $T \in \mathtt{ances}(Y)$, we know $w \in \mathtt{cContent}(t_v,Y) \cup \mathtt{aContent}(t_v,Y)$.

   We can show by contradiction that we must have $w \in \mathtt{aContent}(t_v,Y)$. If $Y = T$, then we already know $w \in \mathtt{aContent}(t_v,Y)$. Otherwise, assume $T \in \mathtt{pAnces}(Y)$. If we had $w \in \mathtt{cContent}(t_v,Y)$, then by Theorem 15, we must have $(\ell,y) \in \mathtt{W}(t_v,Y)$. This statement contradicts the fact that *OAT* model found $(\ell,u)$ from transaction $T$, since a closer transaction $Y$ had $\ell$ in its readset.

But then, since $w \in \texttt{aContent}(t_v, Y)$, we have $wHv$.

2. Suppose $w \in \texttt{vContent}(t_v, T)$:

   Then, we know $w \in \texttt{cContent}(t_v, Y) \cup \texttt{vContent}(t_v, Y)$. As in the previous case, we can show $w \notin \texttt{cContent}(t_v, Y)$.

   If $w \in \texttt{vContent}(t_v, Y)$, then there exists some transaction $Z \in \texttt{activeX}(t_v, Y) - \{Y\}$ such that $\ell \in \texttt{W}(t_v, Z)$.

   Since $w \in \texttt{memOps}(t_v, Z)$, we can strengthen this condition to $Z \in \texttt{activeX}(t_v, \texttt{LCA}(w, v)) - \{\texttt{LCA}(w, v)\}$. This statement leads to a contradiction, however, because $w \in \texttt{W}(t_v, Z)$ must conflict with $v$.

   More formally, by statement Invariant 2 of Theorem 13, any new read operation $v$ at time $t_v$ must satisfy $v \in \texttt{desc}(\texttt{low}(\texttt{writers}(t_v, \ell)))$ ( i.e., $v$ is a descendant of the base of the spine for $\ell$). At time $t_v$, however, we must have $\texttt{low}(\texttt{writers}(t_v, \ell)) \in \texttt{desc}(Z)$.

□

## B. RULES FOR TYPE CHECKING

This appendix contains the type rules for the *OAT* type system. The grammar for the type system is shown below.

$$
\begin{array}{rcl}
P & ::= & defn^*\ e \\
defn & ::= & \textsf{class}\ ocn\langle formal+\rangle\ \textsf{extends}\ oc \\
 & & \textsf{where}\ constr^*\ \{field^*\ meth^*\}\ \mid \\
 & & \textsf{class}\ xcn\langle formal+\rangle\ \textsf{extends}\ xc \\
 & & \textsf{where}\ constr^*\ \{xfield^*\ meth^*\} \\
c & ::= & oc \mid xc \\
oc & ::= & ocn\langle owner+\rangle \mid \textsf{Object}\langle owner\rangle \\
xc & ::= & xcn\langle owner+\rangle \mid \textsf{Xmodule}\langle owner\rangle \\
owner & ::= & \textsf{world}[i] \mid formal \mid \textsf{this}[i] \\
constr & ::= & (owner \rhd owner) \mid (owner \not\rhd owner) \mid \\
 & & (owner = owner) \mid (owner \neq owner) \\
meth & ::= & t\ mn\langle formal^*\rangle(arg^*)\ \textsf{where}\ constr^*\{e\} \\
field & ::= & t\ fd \\
xfield & ::= & c\ fd \\
arg & ::= & t\ x \\
t & ::= & c \mid \textsf{int} \\
formal & ::= & f \\
e & ::= & \textsf{new}\ c \mid x \mid x = e \mid \\
 & & \textsf{let}\ (arg = e)\ \textsf{in}\ \{e\} \mid \\
 & & x.fd \mid x.fd = y \mid x.mn\langle owner^*\rangle(y^*)
\end{array}
$$

$$
\begin{array}{rcl}
ocn & \in & \text{class names that are not subtype of Xmodule} \\
xcn & \in & \text{class names that are subtype of Xmodule} \\
fd & \in & \text{field names} \\
mn & \in & \text{method names} \\
x, y & \in & \text{variable names} \\
f & \in & \text{owner names} \\
i, j & \in & \text{type int literals}
\end{array}
$$

We define a number of predicates used in the type system. These predicates are adapted from [2], but our type system does not handle inner classes for now.

| Predicate | Meaning |
|---|---|
| *WFClasses(P)* | There are no cycles in the class hierarchy |
| *ClassOnce(P)* | No class is declared twice in $P$ |
| *FieldsOnce(P)* | No class contains two fields, decalred or inherited with the same name |
| *MethodsOnce(P)* | No class contains two methods with the same name |
| *OverridesOK(P)* | Overriding methods have the same return type and parameter types as the methods being overridden. |
| *WorldInMainOnly(P)* | Only the main method uses the world tag to initialize owner. |
| *ThisInXcOnly(P)* | Only classes that are subtype of Xmodule use this tag to initialize owner. |

Our typing judgment follows the form adapted from [2]: $P;\ E\ \vdash\ e : t$, where $P$ is the program being checked to provide information about class definitions; $E$ is an environment providing type information for the free variables in $e$; finally, $t$ is the type of $e$.

The typing environment is defined as

$E ::= \emptyset\ \mid\ E, t\ x\ \mid\ E, \text{owner}\ f\ \mid\ E, constr$

The typing environtment contains the the declared types of variables, the decalred owner parameters, the declared constraints among owners, and certain inferred constraints, such as this[i] = this[j] when they are used in a Xmodule class definition.

The typing system uses the following judgments.

| Judgment | Meaning |
|---|---|
| $\vdash\ P : t$ | program $P$ yields type $t$ |
| $P \vdash\ defn$ | $defn$ is a well-formed class |
| $P;\ E\ \vdash\ constr$ | constraint *constr* is satisfied |
| $P;\ E\ \vdash\ (o_1 = o_2)$ | $o_1$ and $o_2$ represent the same owner instance |
| $P;\ E\ \vdash_{\text{owner}}\ o$ | $o$ is an owner |
| $P;\ E\ \vdash\ wf$ | typing environment $E$ is well-formed |
| $P;\ E\ \vdash\ t$ | $t$ is a well-formed type |
| $P;\ E\ \vdash\ t_1 <: t_2$ | $t_1$ is a subtype of $t_2$ |
| $P;\ E\ \vdash\ t_1 <:= t_2$ | $t_2$ is assignable to $t_1$ |
| $P\ \vdash\ xfield \in xc$ | Xmodule class *xc* declares/inherits $xfield$ |
| $P\ \vdash\ field \in oc$ | non-Xmodule class *oc* declares/inherits $field$ |
| $P;\ E\ \vdash\ field$ | $field$ is a well-formed field |
| $P\ \vdash\ meth \in xc$ | Xmodule class *xc* declares/inherits *meth* |
| $P\ \vdash\ meth \in oc$ | non-Xmodule class *oc* declares/inherits *meth* |
| $P;\ E\ \vdash\ meth$ | *meth* is a well-formed method |
| $P;\ E\ \vdash\ e : t$ | expression $e$ has type $t$ |

We present the type rules for these judgments in the following pages.

The type rules for these judgments are presented below:

$\boxed{\vdash P : t}$

[PROG]

$$\frac{WFClasses(P) \quad ClassOnce(P) \quad FieldsOnce(P) \quad MethodsOnce(P) \quad OverridesOK(P) \quad WorldInMainOnly(P) \quad ThisInXcOnly(P) \quad P = defn_{1..n}\ e \quad P \vdash defn_i \quad P;\ \emptyset \vdash e : t}{\vdash P : t}$$

$\boxed{P \vdash defn}$

[CLASS]

$$\frac{E = ocn\langle f_{1..n}\rangle\ \text{this},\ \text{owner}\ f_{1..n},\ f_1 \rhd f_i,\ constr^* \quad P;\ E \vdash wf \quad P;\ E \vdash oc' \quad P;\ E \vdash field_i \quad P;\ E \vdash meth_i}{P \vdash \text{class}\ ocn\langle f_{1..n}\rangle\ \text{extends}\ oc'\ \text{where}\ constr^*\ \{field^*\ meth^*\}}$$

[XMODULE CLASS]

$$\frac{E = xcn\langle f_{1..n}\rangle\ \text{this},\ \text{owner}\ f_{1..n},\ f_1 \rhd f_i,\ constr^* \quad P;\ E \vdash wf \quad P;\ E \vdash xc' \quad P;\ E \vdash field_i \quad P;\ E \vdash meth_i}{P \vdash \text{class}\ xcn\langle f_{1..n}\rangle\ \text{extends}\ xc'\ \text{where}\ constr^*\ \{xfield^*\ meth^*\}}$$

$\boxed{P;\ E \vdash constr}$

[CONSTR ENV] [$\rhd$ WORLD] [$\rhd$ OWNER] [$\rhd$ REFL] [$\rhd$ TRANS]

$$\frac{E = E_1,\ constr,\ E_2}{P;\ E \vdash constr} \qquad \frac{P;\ E \vdash_{\text{owner}}\ o}{P;\ E \vdash (o \rhd \text{world})} \qquad \frac{P;\ E \vdash e : xcn\langle o_{1..n}\rangle}{P;\ E \vdash (e \rhd o_1)} \qquad \frac{P;\ E \vdash_{\text{owner}} o}{P;\ E \vdash (o \rhd o)} \qquad \frac{P;\ E \vdash (o_1 \rhd o_2) \quad P;\ E \vdash (o_2 \rhd o_3)}{P;\ E \vdash (o_1 \rhd o_3)}$$

$\boxed{P;\ E \vdash (o_1 = o_2)}$

[= OWNER] [= REFL] [= TRANS]

$$\frac{E = E_1,\ xc\ \text{this},\ E_2}{P;\ E \vdash (\text{this[i]} = \text{this[j]})} \qquad \frac{P;\ E \vdash_{\text{owner}} o}{P;\ E \vdash (o = o)} \qquad \frac{P;\ E \vdash (o_1 = o_2) \quad P;\ E \vdash (o_2 = o_3)}{P;\ E \vdash (o_1 = o_3)}$$

$\boxed{P;\ E \vdash_{\text{owner}} o}$

[OWNER WORLD] [OWNER FORMAL] [OWNER THIS]

$$\frac{}{P;\ E \vdash_{\text{owner}}\ \text{world}} \qquad \frac{E = E_1,\ \text{owner}\ f,\ E_2}{P;\ E \vdash_{\text{owner}}\ f} \qquad \frac{E = E_1,\ xc\ \text{this},\ E_2}{P;\ E \vdash_{\text{owner}}\ \text{this[i]}}$$

$\boxed{P;\ E \vdash wf}$

[ENV ∅] [ENV X] [ENV OWNER]

$$\frac{}{P;\ \emptyset \vdash wf} \qquad \frac{\begin{array}{c} P;\ E \vdash t \\ x \notin \mathrm{Dom}(E) \\ P;\ E \vdash wf \end{array}}{P;\ E,\ t\ x \vdash wf} \qquad \frac{\begin{array}{c} f \notin \mathrm{Dom}(E) \\ P;\ E \vdash wf \end{array}}{P;\ E,\ \mathrm{owner}\ f \vdash wf}$$

[ENV CONSTR]

$$\frac{\begin{array}{c} constr = (o \rhd o') \ \vee\ (o \not\rhd o') \vee (o = o') \vee\ (o \neq o') \\ P;\ E \vdash wf \qquad P;\ E \vdash_{\mathrm{owner}}\ o,\ o' \qquad E' = E,\ constr \\ \nexists_{x,y}\ (P;\ E' \vdash x \rhd y) \wedge (P;\ E' \vdash x \not\rhd y) \qquad \nexists_{x,y}\ (P;\ E' \vdash x = y) \wedge (P;\ E' \vdash x \neq y) \end{array}}{P;\ E,\ constr \vdash wf}$$

$\boxed{P;\ E \vdash t}$

[TYPE INT] [TYPE OBJECT] [TYPE OC]

$$\frac{}{P;\ E \vdash \mathsf{int}} \qquad \frac{P;\ E \vdash_{\mathrm{owner}}\ o}{P;\ E \vdash \mathsf{Object}\langle o \rangle} \qquad \frac{\begin{array}{c} P \vdash \mathsf{class}\ ocn\langle f_{1..n} \rangle \ldots \mathsf{where}\ constr^* \ldots \\ P;\ E \vdash_{\mathrm{owner}}\ o_i \qquad P;\ E \vdash o_1 \rhd o_i \qquad P;\ E \vdash constr\ [o_1/f_1]..[o_n/f_n] \end{array}}{P;\ E \vdash ocn\langle o_{1..n} \rangle}$$

[TYPE XMODULE] [TYPE XC]

$$\frac{P;\ E \vdash_{\mathrm{owner}}\ o}{P;\ E \vdash \mathsf{Xmodule}\langle o \rangle} \qquad \frac{\begin{array}{c} P \vdash \mathsf{class}\ xcn\langle f_{1..n} \rangle \ldots \mathsf{where}\ constr^* \ldots \\ P;\ E \vdash_{\mathrm{owner}}\ o_i \qquad P;\ E \vdash o_1 \rhd o_i \qquad P;\ E \vdash constr\ [o_1/f_1]..[o_n/f_n] \end{array}}{P;\ E \vdash xcn\langle o_{1..n} \rangle}$$

$\boxed{P;\ E \vdash t_1 <: t_2}$

[SUBTYPE REFL] [SUBTYPE TRANS]

$$\frac{P;\ E \vdash t}{P;\ E \vdash t <: t} \qquad \frac{\begin{array}{c} P;\ E \vdash t_1 <: t_2 \\ P;\ E \vdash t_2 <: t_3 \end{array}}{P;\ E \vdash t_1 <: t_3}$$

[SUBTYPE XC] [SUBTYPE OC]

$$\frac{\begin{array}{c} P;\ E \vdash xcn\langle o_{1..n} \rangle \\ P \vdash \mathsf{class}\ xcn\langle f_{1..n} \rangle\ \mathsf{extends}\ xcn'\langle f_1\ o^* \rangle \ldots \end{array}}{P;\ E \vdash xcn\langle o_{1..n} \rangle\ <:\ xcn'\langle f_1\ o^* \rangle\ [o_1/f_1]..[o_n/f_n]} \qquad \frac{\begin{array}{c} P;\ E \vdash ocn\langle o_{1..n} \rangle \\ P \vdash \mathsf{class}\ ocn\langle f_{1..n} \rangle\ \mathsf{extends}\ ocn'\langle f_1\ o^* \rangle \ldots \end{array}}{P;\ E \vdash ocn\langle o_{1..n} \rangle\ <:\ ocn'\langle f_1\ o^* \rangle\ [o_1/f_1]..[o_n/f_n]}$$

$\boxed{P;\ E \vdash t_1 <:= t_2}$

[ASSIGNABILITY REFL]

$$\frac{P;\ E \vdash t}{P;\ E \vdash t <:= t}$$

[ASSIGNABILITY TRANS]

$$\frac{P;\ E \vdash t_1 <:= t_2 \qquad P;\ E \vdash t_2 <:= t_3}{P;\ E \vdash t_1 <:= t_3}$$

[ASSIGNABILITY FOR XC]

$$\frac{P;\ E \vdash xcn\langle o_{1..n}\rangle \qquad P;\ E \vdash xcn\langle o'_{1..n}\rangle \qquad P;\ E \vdash (o_i = o'_i)^{i\in 1..n} \qquad P;\ E \vdash (o_i \rhd o'_i)^{i\in 1..n}}{P;\ E \vdash xcn\langle o_{1..n}\rangle <:= xcn\langle o'_{1..n}\rangle}$$

[ASSIGNABILITY FOR OC]

$$\frac{P;\ E \vdash ocn\langle o_{1..n}\rangle \qquad P;\ E \vdash ocn\langle o'_{1..n}\rangle \qquad P;\ E \vdash (o_i = o'_i)^{i\in 1..n} \qquad P;\ E \vdash (o_i \rhd o'_i)^{i\in 1..n}}{P;\ E \vdash ocn\langle o_{1..n}\rangle <:= ocn\langle o'_{1..n}\rangle}$$

$\boxed{P \vdash xfield \in xc}$

[XFIELD DECLARED]

$$\frac{P \vdash \mathsf{class}\ xcn\langle f_{1..n}\rangle ... \{... xfield ...\}}{P \vdash xfield \in xcn\langle f_{1..n}\rangle}$$

[XFIELD INHERITED]

$$\frac{P \vdash xfield \in xcn\langle f_{1..n}\rangle \qquad P \vdash \mathsf{class}\ xcn'\langle g_{1..m}\rangle\ \mathsf{extends}\ xcn\langle o_{1..n}\rangle ...}{P \vdash xfield\ [o_1/f_1]..[o_n/f_n] \in xcn'\langle g_{1..m}\rangle}$$

$\boxed{P \vdash field \in oc}$

[FIELD DECLARED]

$$\frac{P \vdash \mathsf{class}\ ocn\langle f_{1..n}\rangle ... \{... field ...\}}{P \vdash field \in ocn\langle f_{1..n}\rangle}$$

[FIELD INHERITED]

$$\frac{P \vdash field \in ocn\langle f_{1..n}\rangle \qquad P \vdash \mathsf{class}\ ocn'\langle g_{1..m}\rangle\ \mathsf{extends}\ ocn\langle o_{1..n}\rangle ...}{P \vdash field\ [o_1/f_1]..[o_n/f_n] \in ocn'\langle g_{1..m}\rangle}$$

$\boxed{P;\ E \vdash field}$

[FIELD]

$$\frac{P;\ E \vdash t}{P;\ E \vdash t\ fd}$$

$\boxed{P \vdash meth \in xc}$

[METHOD DECLARED IN XC]

$$\frac{P \vdash \mathsf{class}\ xcn\langle f_{1..n}\rangle ... \{... meth ...\}}{P \vdash meth \in xcn\langle f_{1..n}\rangle}$$

[METHOD INHERITED BY XC]

$$\frac{P \vdash meth \in xcn\langle f_{1..n}\rangle \qquad P \vdash \mathsf{class}\ xcn'\langle g_{1..m}\rangle\ \mathsf{extends}\ xcn\langle o_{1..n}\rangle ...}{P \vdash meth\ [o_1/f_1]..[o_n/f_n] \in xcn'\langle g_{1..m}\rangle}$$

$\boxed{P \vdash meth \in oc}$

[METHOD DECLARED IN OC]

$$\frac{P \vdash \mathsf{class}\ ocn\langle f_{1..n}\rangle ... \{... meth ...\}}{P \vdash meth \in ocn\langle f_{1..n}\rangle}$$

[METHOD INHERITED BY OC]

$$\frac{P \vdash meth \in ocn\langle f_{1..n}\rangle \qquad P \vdash \mathsf{class}\ ocn'\langle g_{1..m}\rangle\ \mathsf{extends}\ ocn\langle o_{1..n}\rangle ...}{P \vdash meth\ [o_1/f_1]..[o_n/f_n] \in ocn'\langle g_{1..m}\rangle}$$

$\boxed{P;\ E\ \vdash\ meth}$

[METHOD]

$$\frac{E' = E,\ \text{owner}\ f_{1..n},\ constr^*,\ arg^* \qquad P;\ E'\ \vdash\ wf \qquad P;\ E'\ \vdash\ e : t}{P;\ E\ \vdash\ t\ mn\langle f_{1..n}\rangle(arg^*)\ \text{where}\ constr^*\ \{e\}}$$

$\boxed{P;\ E\ \vdash\ e : t}$

[EXP SUB]

$$\frac{P;\ E\ \vdash\ e : t' \qquad P;\ E\ \vdash\ t'\ <:\ t}{P;\ E\ \vdash\ e : t}$$

[EXP NEW]

$$\frac{P;\ E\ \vdash\ c}{P;\ E\ \vdash\ \text{new}\ c : c}$$

$\boxed{P;\ E\ \vdash\ e : t}$

[EXP ASSIGNABILITY]

$$\frac{P;\ E\ \vdash\ e : t' \qquad P;\ E\ \vdash\ t'\ <:=\ t}{P;\ E\ \vdash\ e : t}$$

[EXP LET]

$$\frac{arg = t\ x \qquad P;\ E\ \vdash\ e : t \qquad P;\ E,\ arg\ \vdash\ e' : t'}{P;\ E\ \vdash\ \text{let}\ (arg = e)\ \text{in}\ \{e'\} : t'}$$

[EXP VAR]

$$\frac{E = E_1,\ t\ x,\ E_2}{P;\ E\ \vdash\ x : t}$$

[EXP VAR ASSIGN]

$$\frac{P;\ E\ \vdash\ x : t \qquad P;\ E\ \vdash\ e : t}{P;\ E\ \vdash\ x = e : t}$$

[EXP REF]

$$\frac{P;\ E\ \vdash\ x : cn\langle o_{1..n}\rangle \qquad P\ \vdash\ (t\ fd)\ \in\ cn\langle f_{1..n}\rangle}{P;\ E\ \vdash\ x.fd : t\ [o_1/f_1]..[o_n/f_n]}$$

[EXP REF ASSIGN]

$$\frac{P;\ E\ \vdash\ x : cn\langle o_{1..n}\rangle \qquad P\ \vdash\ (t\ fd)\ \in\ cn\langle f_{1..n}\rangle \qquad P;\ E\ \vdash\ y : t\ [o_1/f_1]..[o_n/f_n]}{P;\ E\ \vdash\ x.fd\ = y : t\ [o_1/f_1]..[o_n/f_n]}$$

[EXP INVOKE]

$$\frac{\begin{array}{c}P\ \vdash\ (t\ mn\langle f_{(n+1)..m}\rangle(t_j\ y_j\ ^{j\in 1..k})\ \text{where}\ constr^*\ ...)\ \in\ cn\langle f_{1..n}\rangle \\ P;\ E\ \vdash\ x : cn\langle o_{1..n}\rangle \qquad P;\ E\ \vdash\ x_j : t_j\ [o_1/f_1]..[o_m/f_m] \\ P;\ E\ \vdash\ o_1\ \rhd\ o_i \qquad P;\ E\ \vdash\ constr\ [o_1/f_1]..[o_m/f_m]\end{array}}{P;\ E\ \vdash\ x.mn\langle o_{(n+1)..m}\rangle(x_{1..k}) : t\ [o_1/f_1]..[o_m/f_m]}$$