Safe Open-Nested Transactions Through Ownership

Kunal Agrawal   I-Ting Angelina Lee   Jim Sukha
MIT Computer Science and Artificial Intelligence Laboratory
{kunal_ag, angelee, sukhaj}@mit.edu

ABSTRACT

Researchers in transactional memory (TM) have proposed open nesting as a methodology for increasing the concurrency of transactional programs. The idea is to ignore "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 the 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, unconstrained use of open-nested commits can lead to anomalous program behavior.

We describe the 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 which Xmodule owns the data. Moreover, we provide a set of precise constraints on interactions and sharing of data among the Xmodules based on familiar notions of abstraction. The ownership-aware commit mechanism and these restrictions on Xmodules allow us to prove that ownership-aware TM has clean memory-level semantics. In particular, it guarantees ***serializability by modules***, an adaptation of the definition of multilevel serializability from database systems. 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, then ownership-aware TM is free from ***semantic deadlock***.

1. INTRODUCTION

Transactional memory (TM) [6] 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 this section of code executes atomically. A TM system enforces atomicity by tracking the memory locations that each transaction accesses (using ***read sets***

This research was supported in part by NSF Grants NSF Grants CNS-0615215 and CNS-0540248 and a grant from Intel corporation.
Preliminary versions of this paper appeared as a poster at ***PPoPP*** 2008 and as a brief announcement at ***SPAA*** 2008.

and ***write sets***), finding transaction conflicts, and aborting transactions that conflict. TM guarantees that transactions are ***serializable*** [12]; 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 issue that programmers must deal with is the semantics of ***nested transactions***. If a transaction Y is ***closed nested*** [8] inside transaction X, for the purpose of detecting conflicts, the TM system considers any memory locations accessed by Y as conceptually also being accessed by its parent X. Thus, upon Y's commit, the TM system merges Y's read and write sets into the read and write sets of 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 programs because it does not allow two "high-level" transactions to ignore conflicts due to accesses to shared "low-level" memory done by nested transactions.

Researchers have proposed the methodology of ***open-nested transactions*** to increase concurrency in transactional programs by carefully breaking serializability at the memory level. The open-nesting methodology incorporates the ***open-nested commit mechanism*** [7, 10]. When an open-nested transaction Y (nested inside transaction X) commits, Y's changes are committed to memory and Y's read and write sets are discarded. Thus, the TM system 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; thus 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, if X aborts, it may need to execute ***compensating actions*** to undo the effect of its committed open-nested subtransaction Y. Moss in [9] illustrates use of open nesting with an application that uses a B-tree. Ni et al. [11] 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 that can be tricky to reason about [2]. We believe that one reason for the apparent complexity of open nesting is that the mechanism and the 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 ℓ_1 and ℓ_2, and X does not care about changes made to ℓ_2, but does care about ℓ_1. The TM system can not distinguish between these two accesses, and will commit both in an open-nested manner, leading to anomalous behavior.

Researchers *have* demonstrated specific examples [4, 11] that safely use an open-nested commit mechanism. These examples work, however, because the inner (open) transactions never write to any data that is accessed by the outer transactions. Moreover, since these 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. 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.

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 the ***ownership-aware TM system*** allows the programmer to safely use the methodology of open nesting, because the runtime's behavior more closely reflects the programmer's intent. In addition, the 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 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 most of the above-mentioned guidelines in the programs written using this language extension.
3. We formally describe the operational model for ownership-aware TM, called the *OAT* model, which uses a new ***ownership-aware commit mechanism***. The ownership-aware commit mechanism is a compromise between an open-nested and a closed-nested commit; when a transaction T commits, a change to memory location ℓ is committed globally if ℓ belongs to the module of T; otherwise, the read or write to ℓ is propagated to T's parent transaction. Unlike an ordinary open-nested commit, the ownership-aware commit treats memory locations differently depending on which Xmodule 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 implement the full methodology.
4. We prove that if a program follows the proposed guidelines for 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. Thus, one corollary of our theorem is that open-nested transactions are serializable when modules do not share data. This observation explains why researchers [4, 11] have found it natural to use open-nested transactions in the absence of sharing, in spite of the apparent semantic pitfalls.
5. We prove that under certain restricted conditions, a computation executing under the *OAT* model can not enter a semantic deadlock.

In later sections, we distinguish between the variations of nested transactions as follows. We say that a transaction Y is ***vanilla open nested*** when referring to a TM system which performs the open-nested commit of Y. We say that Y is ***safe nested*** when referring to the ownership-aware TM system which performs the ownership-aware commit of Y. Finally, we say that a transaction Y is an open-nested transaction when we are referring to the abstract methodology, rather than a particular implementation with a specific commit mechanism.

Outline

The paper is organized as follows. In Section 2 we present an overview of ownership-aware TM and highlight key features using an example application. Section 3 describes language constructs for specifying Xmodules and ownership. In Section 4, we review the transactional computation framework [2], and extend this framework to formally incorporate Xmodules and ownership. Section 5 describes the *OAT* model, and Section 6 gives a formal definition of serializability by modules, and a proof-sketch that the *OAT* model guarantees this definition. Section 7 provides conditions under which the *OAT* model does not exhibit semantic deadlocks. Section 8 concludes with a discussion of some related work.

2. OWNERSHIP-AWARE TRANSACTIONS

In this section, we give an overview of ownership-aware TM. To motivate the need for the concept of ownership in TM, we first present an example application which might benefit from open nesting. We then introduce the notion of an Xmodule and informally explain the programming guidelines when using Xmodules. Finally, we highlight some of the key differences between ownership-aware TM and a TM with vanilla open nesting. In this section, we present the intuitive descriptions of the concepts in ownership-aware TM; we defer formal definitions until later sections.

Example Application

We describe an example application for which one might use open-nested transactions. This example is similar to the one in [9], but it includes data sharing between nested transactions and their parents, and has more than two levels of nesting.

Since the open-nesting methodology is designed programs to have multiple levels of abstraction, we choose a modular application. Consider a user application which concurrently accesses a database of many individuals' book collections. The database stores records in a binary search tree, keyed by name. Each node in the binary search tree corresponds to a person, and stores a list of books in his/her 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; given a book title, it returns a list of people who own the book. Finally, the user application wants the database to log changes on disk for recoverability. Whenever the database is updated, it 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.

This application is modular, with five natural modules — the user application (`UserApp`), the database (`DB`), the binary search tree (`BST`), the hashtable (`Hashtable`), and the logger (`Logger`). The `UserApp` module calls methods from the `DB` module when it wants to insert into the database, or query the database. The database in turn maintains internal metadata and calls the `BST` module and the `Hashtable` module to answer queries and insert data. Both user application and the database may call methods from the `Logger` module.

If the modules use open-nested transactions, a TM system with vanilla open-nested commits can result in non-intuitive outcomes. Consider the example where a transactional method A from the `UserApp` module tries to insert a book b into the database, and the insert is an open-nested transaction. The method A (which corresponds to transaction X) calls an insert method in the `DB` module and passes b (the `Book` object) to be inserted. This insert method generates an open-nested transaction Y. Suppose Y writes to some field of the book b (memory location ℓ_1), and also writes some internal database metadata (location ℓ_2). After a vanilla open-nested commit of Y, the modifications to both ℓ_1 and ℓ_2 become visible globally. Assuming the `UserApp` does not care about the internal state of the database, committing the internal state of the `DB` (ℓ_2) is a desirable effect of open nesting; this commit increases concurrency, because other transactions can potentially modify the database in parallel with X without generating a conflict. The `UserApp` does, however, care about changes to the book b; thus, the commit of ℓ_1 breaks the atomicity of transaction X. A transaction Z in parallel with transaction X can access this location ℓ_1 after Y commits, before the outer transaction X commits.[1] To increase concurrency, we want the method from `DB` to commit changes to its own internal data; we do not, however, want it to commit the data that `UserApp` cares about.

To enforce this kind of restriction, we need some notion of ***ownership of data***: if the TM system is aware of the fact that the book object "belongs" to the `UserApp`, then it can decide not to commit `DB`'s change to the book object globally. For this purpose, we introduce the notion of ***transactional modules***, or Xmodules. When a programmer explicitly defines Xmodules and specifies the ownership of data, the TM system can make the correct judgement about which data to commit globally.

Xmodules and the Ownership-Aware Commit Mechanism

The ownership-aware TM system requires that programs be organized into Xmodules. Intuitively, an Xmodule M is as a stand-alone entity that contains data and transactional methods; an Xmodule owns data that it privately manages, and uses its methods to provide public services to other modules. During program execution, a call to a method from Xmodule M generates a transaction instance (e.g., X). If this method in turn calls another method from an Xmodule N, an additional transaction Y, safe nested inside X, is created only if $M \neq N$. Therefore, defining an Xmodule automatically specifies safe-nested transactions.

[1] Note that abstract locks [9] do not address this problem. Abstract locks are meant to disallow other transactions from noticing the fact that the book was inserted into the `DB`. They do not usually protect the individual fields of the book object itself.

In the ownership-aware TM system, every memory location is owned by exactly one Xmodule. If a memory location ℓ is in a transaction T's read or write set, the ownership-aware commit of a transaction T commits this access globally only if T is generated by the same Xmodule that owns ℓ; in this case, we say that T is "responsible" for that access to ℓ. Otherwise, the read or write to ℓ is propagated up to the read or write set of T's parent transaction; that is, the TM system behaves as though T was a closed-nested transaction with respect to location ℓ.

For ownership-aware TM to behave "nicely", we must restrict interactions between Xmodules. For example, in the TM system, some transaction must be "responsible" for committing every memory access. Similarly, the TM system should guarantee some form of serializability. If Xmodules could arbitrarily call methods from or access memory owned by other Xmodules, then these two properties might not be satisfied.

Rules for Xmodules

Ownership-aware TM uses Xmodules to control both the structure of nested transactions, and the sharing of data between Xmodules (i.e., to limit which memory locations a transaction instance can access). In our system, Xmodules are arranged as a ***module tree***, denoted as $\mathcal{D}$. In $\mathcal{D}$, an Xmodule N is a child of M if N is "encapsulated by" M. The root of $\mathcal{D}$ is a special Xmodule called `world`. Each Xmodule is assigned an `xid` by visiting the nodes of $\mathcal{D}$ in a left-to-right depth-first search order, and assigning ids in increasing order, starting with $\texttt{xid}(\texttt{world}) = 0$. Therefore `world` has the minimum `xid`, and "lower-level" Xmodules have larger `xid` numbers.

DEFINITION 1. *We impose two rules on Xmodules based on the module tree:*

1. **Rule 1***: A method of an Xmodule M can access a memory location ℓ directly only if ℓ is either owned by M or an ancestor of M in the module tree. This rule means that an ancestor Xmodule N of M may pass data down to a method belonging to M, but a transaction from module M can not directly access any "lower-level" memory.*
2. **Rule 2***: A method from M can call a method from N only if N is the child of some ancestor of M, and $\texttt{xid}(N) > \texttt{xid}(M)$ (i.e., if N is "to the right" of M in the module tree). This rule requires that an Xmodule can call methods of some (but not all) lower-level Xmodules.*[2]

The intuition behind these rules is as follows. Xmodules have methods to provide services to other higher-level Xmodules, and Xmodules maintain their own data in order to provide these services. Therefore, a higher-level Xmodule can pass its data to a lower-level Xmodule and ask for services. A higher-level Xmodule should not directly access the internal data belonging to a lower-level Xmodule.

If Xmodules satisfy Rules 1 and 2, TM can have a well-defined ownership-aware commit mechanism; some transaction is always "responsible" for every memory access (proved in Section 5). In addition, these rules and the ownership-aware commit mechanism guarantee that transactions satisfy the property of "serializability by modules" (proved in Section 6).

One potential limitation of ownership-aware TM is that some "cyclic dependencies" between Xmodules are prohibited. The ability to define one module as being at a lower level than another is fundamental to the open-nesting methodology. Thus, our formalism requires that Xmodules be partially ordered; if an Xmodule M can call Xmodule N, then conceptually M is at a higher level than N (i.e., $\texttt{xid}(M) < \texttt{xid}(N)$), and thus N can not call M. If two components of the program call each other, then, conceptually, neither of these components is at a higher-level than the other, and we would require that these two components be combined into the same Xmodule.

[2] An Xmodule can, in fact, call methods within its own Xmodule or from its ancestor Xmodules, but we model these calls differently. We explain these cases condition at the end of this section.

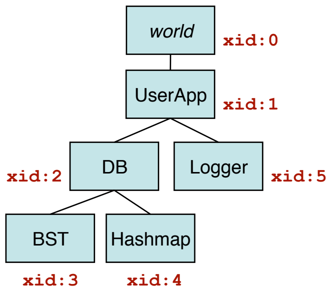

Figure 1. A module tree $\mathcal{D}$ for the program described in Section 2. The `xid`'s are assigned according to a left-to-right depth-first tree walk, numbering Xmodules in increasing order, starting with `xid`(`world`) = 0.

Xmodules in the Example Application

Consider a Java implementation of the example application described earlier. It may have 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: tree node `BSTNode` for the `BST`, `Bucket` in the `Hashmap`, and `Buffer` used by the `Logger`.

For ownership-aware TM, not all of a program's classes are meant to be Xmodules; some classes only wrap data. In our example, we identified five Xmodules– `UserApp`, `DB`, `BST`, `Hashmap`, and `Logger`; these classes are stand-alone entities which have encapsulated data and methods. 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 their internal state.

We organize the Xmodules of the application into the module tree shown in Figure 1. `UserApp` is encapsulated by `world`, `DB` and `Logger` are encapsulated under `UserApp`; `BST` and `Hashmap` are encapsulated under `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 by either `UserApp` or its descendants.

Let us consider the implications of Definition 1 for the example. Due to Rule 1, all of `DB`, `BST`, `Hashmap`, and `Logger` can 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` should be able to pass its data down, allowing lower-level Xmodules to access that data directly, but `UserApp` itself should not be able to directly access data owned by lower-level Xmodules. Due to Rule 2, the `UserApp` may invoke methods from `DB`, `DB` may invoke methods from `BST` and `Hashmap`, and every other Xmodule may invoke methods from `Logger`. Thus, Rule 2 allows all the operations required by the example application. As expected, the `UserApp` can call the `insert` and `search` methods from the `DB` and can even pass its data to the `DB` for insertion. More importantly, notice the relationship between `BST` and `Logger`. The `BST` Xmodule can call methods from `Logger`, but the `BST` can not pass data it owns directly into the `Logger`. It can, however, pass data owned by the `UserApp` to the logger, which is all this application requires.

Advantage of Ownership-Aware Transactions

One of the major problems with vanilla open nesting is that some transactions can see inconsistent data. Say a transaction Y is open nested inside transaction X. Let v_0 be the initial value of location ℓ, and suppose Y writes value v_1 to location ℓ and then commits. Now a transaction Z in parallel with X can read this location ℓ, write value v_2 to ℓ, and commit, all before X commits. Therefore, X can now read this location ℓ and see the value v_2, which is neither the initial value v_0 (the value of ℓ when X started), nor v_1 which was written by X's inner transaction, Y. This behavior might seem counterintuitive.

Now consider the same example for ownership-aware transactions. Say X is generated by a method of Xmodule M and Y is generated by a method of Xmodule N. If N owns ℓ, X can not access ℓ, since $\texttt{xid}(M) < \texttt{xid}(N)$ (by Definition 1, Rule 2), and no transaction from a higher-level module can access data owned by a lower-level module (by Definition 1, Rule 1). Thus, the problem does not arise. If N does not own ℓ, the ownership-aware commit of Y will not commit the changes to ℓ globally and ℓ will be propagated to X's write set. Therefore, if Z tries to access ℓ before X commits, the TM system will detect a conflict. Thus X can not see an inconsistent value for ℓ.[3]

Callbacks

At first glance, the assumptions we have made regarding methods of Xmodules seem somewhat restrictive. In the description thus far, we prohibit an Xmodule M from calling another transactional method from M or a proper ancestor of M. In particular, it appears as though our model disallows callbacks. Our model, however, does permit both these cases; we simply model these calls differently.

If a method X from Xmodule M calls another method Y from an ancestor Xmodule N, this new call does not generate a new safe-nested transaction instance. Instead, Y is subsumed in X using flat (or closed) nesting. Recall that Rule 1 in Definition 1 allows X to access data belonging to N or any of its ancestors directly. Therefore, we can treat any data access by a flat (or closed) nested transaction Y as being accessed by X directly, provided that Y and its nested transactions access only memory belonging to N or N's ancestors. We say that Y is a ***proper callback*** method for Xmodule N if its nested calls are all proper callback methods belonging to Xmodules which are ancestors of N. In our formal model in Section 4, we assume that we only have proper callbacks and model them as direct memory accesses, allowing us to ignore them in the formal definitions.

Closed-Nested Transactions

In our model, every method call that crosses an Xmodule boundary automatically generates a safe-nested transaction. Ownership-aware TM can effectively provide closed-nested transactions, however, with appropriate specifications of ownership. If an Xmodule M owns no memory, but only operates on memory belonging to its proper ancestors, then transactions of M will effectively be closed nested. In the limit, if the programmer specifies that all memory is owned by the `world` Xmodule, then all changes in any transaction's read or write set are propagated upwards; thus all ownership-aware commits behave exactly as closed-nested commits.

3. OWNERSHIP TYPES FOR Xmodules

When using ownership-aware transactions, the Xmodules and data ownership in a program must be specified for two reasons. First, the ownership-aware commit mechanism depends on these concepts. Second, we can guarantee some notion of serializability only if a program has Xmodules which conform to the rules in Definition 1. In this section, we describe language constructs and a type system that can be used to specify Xmodules and ownership in a Java-like language. Our type system — the *OAT* ***type system*** — statically enforces some of the restrictions described in Definition 1.

The *OAT* type system extends the ownership types of Boyapati et al. [3], which is described first in this section. We then describe extensions to this type system to enforce some of the restrictions in Definition 1. Next, we present code for parts of the example application described in Section 2. Finally, we discuss some restrictions required by Definition 1 which the *OAT* type system does not enforce statically. The type system's annotations, however, enable dynamic checks for these restrictions.

[3] For simplicity, we have described the case where Y is directly nested inside X. The case where Y is more deeply open nested inside X behaves in a similar fashion.

Boyapati et al.'s Parametric Ownership Type System

The type system of Boyapati et al. provides a mechanism for specifying ownership of objects. The type system enforces the properties stated in Lemma 1.

LEMMA 1. *The type system in [3] 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 *(of objects) rooted at* `world`.
4. *The owner of an object does not change over time.*
5. *An object a can access another object b directly only if b's owner is either a, or one of a's proper ancestors in the ownership tree.*

Boyapati et al.'s type system requires ownership annotations to class definitions and type declarations to guarantee Lemma 1. Every class type `T1` has a set of associated ownership tags, denoted `T1`$\langle f_1, f_2, \ldots f_n\rangle$. The first formal f_1 denotes the owner of the current instance of the object (i.e., `this` object). The remaining formals $f_2, f_3, \ldots f_n$ are additional tags which can be used to instantiate and declare other objects within the class definition. The formals get assigned with actual owners $o_1, o_2, \ldots o_n$ when an object a of type `T1` is instantiated. By parameterizing class and method declarations with ownership tags, the type system of [3] permits owner polymorphism. Thus, one can define a class type (e.g. a generic hash table) once, but instantiate multiple instances of that class with different owners in different parts of the program.

The type system enforces the properties in Lemma 1 by performing the following checks:

1. Within the class definition of type `T1`, only the tags $\{f_1, f_2, \ldots f_n\} \cup \{$`this`,`world`$\}$ are visible. The `this` ownership tag represents the object itself.
2. Within a class definition, a variable c_2 with type `T2`$\langle f_2, \ldots\rangle$ can be assigned to a variable c_1 with type `T1`$\langle f_1, \ldots\rangle$ if and only if `T2` is a subtype of `T1` and $f_1 = f_2$.
3. If an object a's tags are instantiated to be $o_1, o_2, \ldots o_n$ when a is created, then in the ownership tree, o_1 must be a descendant of o_i, $\forall i \in 2..n$, (denoted by $o_1 \preceq o_i$ henceforth).

It is shown in [3] that these type checks guarantee the properties of Lemma 1.

In some cases, to enable the type system to perform check 3 locally, the programmer may need to specify a `where` clause in a class declaration. For example, suppose the class declaration of type `T1` has formal tags $\langle f_1, f_2, f_3\rangle$, and inside `T1`'s definition, some type `T2` object is instantiated with ownership tags $\langle f_2, f_3\rangle$. The type system can not determine whether or not $f_2 \preceq f_3$. To resolve this ambiguity, the programmer must specify `where (`f_2 `<=` f_3`)` at the class declaration of type `T1`. When an instance of type `T2` object is instantiated, the type system then checks that the `where` clause is satisfied.

The* OAT *Type System

The ownership tree described in [3] exhibits some of the same properties as the module tree we described in Section 2; however, the type system and ownership scheme of [3] do not enforce two major requirements of our system.

- In [3], any object can own other objects. Our rules, however, require that only Xmodules own other objects.
- In [3], an object can call any of its ancestor's siblings. Our rules (namely Definition 1), however, dictate that an Xmodule M can only call its ancestor's siblings to the right.

With these requirements in mind, we extend Boyapati et al.'s type system to create the *OAT* type system.

The extensions to handle the first requirement are straightforward. The *OAT* type system explicitly distinguishes objects and Xmodules by requiring that Xmodules extend from a special `Xmodule` class. The

OAT type system only allows classes that extend `Xmodule` to use `this` as an ownership tag. In the context of the Boyapati et al.'s ownership tree, this restriction creates a tree where all the internal nodes are Xmodules and all leaves are non-Xmodule objects. If we ignore any order imposed on the children of an Xmodule, for ownership-aware TM, the module tree (as described in Section 2) is essentially the ownership tree with all non-Xmodule objects removed.

The second requirement is more complicated to enforce. First, we extend each owner instance o to have two fields: ***name***, represented by $o.name$; and ***index***, represented by $o.index$. The name field is conceptually the same as an ownership instance in the type system of [3]. The index field is added to help the compiler to infer ordering between children of the same Xmodule in the module tree. The *OAT* type system allows the programmer to pass `this[i]` as the ownership tag (i.e., with an index i) instead of `this`. Similarly, one can use `world[i]` as an ownership tag. Indices enable the type system to infer an ordering between two sibling Xmodules M and N; for instance, if an Xmodule L instantiates M and N with owners `this[i]` and `this[i+1]`, respectively, then M appears to the left of N in the module tree.

Finally, for technical reasons, the *OAT* system prohibits all Xmodules M from declaring primitive fields. If M had primitive fields, then by Boyapati et al.'s type system, these fields are owned by the M's parent. Since this property seems counter-intuitive, we opted to disallow primitive fields for Xmodules.

In summary, the *OAT* type system performs these checks:

1. Within the class definition of type `T1`, only the tags $\{f_1, f_2, \dots f_n\} \cup \{\texttt{this}, \texttt{world}\}$ are visible.
2. In a class declaration, a variable c_2 with type $\texttt{T2}\langle f_2, \dots\rangle$ can be assigned to a variable c_1 with type $\texttt{T1}\langle f_1, \dots\rangle$ if and only if `T2` and `T1` have the same type and all the formals match in name. In addition, if the indices are specified for the tags, then they must match.
3. For a type $\texttt{T}\langle o_1, o_2, \dots o_n\rangle$, we must have, for all $i \in \{2, \dots n\}$, either $o_1.name \prec o_i.name$ or $o_1.name = o_i.name$ and $o_1.index < o_i.index$ (if both indices are known).[4]
4. The ownership tag `this` can only be used within the definition of a class that extends `Xmodule`.
5. Xmodule objects can not have primitive-type fields.

The first three checks are analogous to the checks in Boyapati et al.'s type system. The last two checks are added to enforce the additional requirements of Xmodules.

The *OAT* type system supports `where` clauses of the form `where` $(f_i < f_j)$; when f_i and f_j are instantiated with o_i and o_j, the type system ensures that either $o_i.name \prec o_j.name$, or $o_i.name = o_j.name$ and $o_i.index < o_j.index$. The detailed type rules for the *OAT* type system are described in [1].

Example Application using the *OAT* Type System

Figure 2 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 `dataO` which is the owner of 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 11–13 illustrate how the `DB` class can initialize its Xmodules and propagate the formal owner tags (i.e., `logO` and `dataO`) down.

Note that in order for this code to type check, the `DB` class must declare `logO < dataO` using the `where` clause in line 10, otherwise the type check would fail at line 11, due to ambiguity of their relation in the module tree. The `where` clause in line 10 is checked whenever an instance of `DB` is created, i.e. at line 6.

[4] In the ownership tree, for any Xmodule M, the *OAT* type system implicitly assigns non-Xmodule children of M higher indices than the Xmodule children of M, unless the user specifies otherwise.

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

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

Figure 2. Specifying Xmodules and ownership for the example application described in Section 2.

The OAT *Type System's Guarantees*

The following lemma about the *OAT* type system can be proved in a reasonably straightforward manner using Lemma 1.

LEMMA 2. *The OAT type system guarantees the following properties.*

1. *An Xmodule M can access a (non-Xmodule) object b with ownership tag o_b only if $M \preceq o_b.name$.*
2. *An Xmodule M can call a method in another Xmodule N with owner o_N only if one of the following is true:*
   (a) *$M = o_N.name$ (i.e. M owns N);*
   (b) *The least common ancestor of M and N in the module tree is $o_N.name$.*
   (c) *$N \succeq M$ (i.e. N is an ancestor of M).*

Lemma 2 does not, however, guarantee all the properties we want from Xmodules (i.e., Definition 1). In particular, Lemma 2 does not consider any ordering of sibling Xmodules. The *OAT* type system can, however, provide stronger guarantees for a program which satisfies what we call the ***unique owner indices*** assumption: for all Xmodules M, all children of M in the module tree are instantiated with ownership tags with unique indices that can be statically determined. For such a program, the type system can order the children of every Xmodule M from smallest to largest index, and assign the `xid` to each Xmodule as described in Section 2. Then, the following result holds:

THEOREM 3. *For a program with unique owner indices, in addition to Lemma 2, the OAT type system guarantees that if the least common ancestor of Xmodules M and N in the module tree is $o_N.name$, then M can call a method in N only if $\texttt{xid}(M) < \texttt{xid}(N)$.*

PROOF.

We prove (by contradiction) that if least common ancestor of M and N in the module tree is $o_N.name$, and $\texttt{xid}(M) > \texttt{xid}(N)$, then M can not have a formal tag with value o_N. Therefore, it can not declare a type with owner tag o_N, and can not access N.

Let L be the least common ancestor of M. Since $L = o_N.name$, we know that L is N's parent. Let Q be the ancestor of M which is N's sibling, and let o_Q be Q's ownership tag (i.e., the tag with which Q is instantiated).

Since N and Q have the same parent (i.e. L) in the module tree, we have $o_N.name = o_Q.name = L$. Since $\texttt{xid}(M) > \texttt{xid}(N)$, M is to the right of N in the ownership tree. Therefore, Q, which is an ancestor of M, is to the right of N in the ownership tree. Therefore, we have $o_Q.index > o_N.index$.

Assume for contradiction that M does have o_N as one of its tags. Using Lemma 1, one can show that the only way for M to receive tag o_N is if Q also has a formal tag with value o_N. Thus, Q's first formal owner tag has value o_Q and another one of its formals has value o_N.

Let $P_0 = Q$, and consider the chain of Xmodule instantiations where Xmodule P_i instantiated P_{i-1}. P_1 has to instantiate Q (which is the same as P_0) using its formal ownership tags $\langle f_a^1, f_b^1, \ldots \rangle$, where f_a^1 has value o_Q and f_b^1 has value o_N. (We must have f_a^1 as the first formal, since o_Q is the owner of Q. Without loss of generality, we can have f_b^1 be the second formal since the type system does not care about the ordering of formal tags after the first one.)

Since $o_N.name = o_Q.name = L$, this chain of instantiations must lead back to L, since that is the only Xmodule that can create ownership tags with values o_N and o_Q in its class definition (using the keyword `this`). [5] Let $P_k = L$. For the class declaration of each of the Xmodules P_i for $1 \le i < k$, the following must be true.

- P_i must have formals f_a^i and f_b^i, with values o_Q and o_N, respectively, and P_i must pass these formals into the instantiation of P_{i-1}.
- In the type definition of P_i's class, P_i must have the constraint $f_a^i < f_b^i$ on its formal tags (either because f_a^i is the owner tag, or through a `where` clause that enforces $f_a^i < f_b^i$.

ssible?

The first condition must hold for us to be able to pass both o_N and o_Q down to $P_0 = Q$. The second condition is true for the Xmodules by induction. In the base case, P_1 must know that $f_a^1 < f_b^1$; otherwise, the type system will throw an error when it tries to instantiate $P_0 = Q$ with owner f_a^1. Then, inductively, P_i must know $f_a^i < f_b^i$ to be able to instantiate P_{i-1}.

Finally, P_{k-1} is instantiated in the class file corrsponding to $P_k = L$. In this declaration, the formal f_a^k with value o_Q is instantiated with $\texttt{this}[x]$. Similarly, f_b^k with value o_N is instantiated with $\texttt{this}[y]$. Since the class definition of P_k type checks, we must have $f_a^k < f_b^k$. This check contradicts our original assumption that $x > y$ however, since if $x > y$ our type check should fail. Therefore, we must have $o_Q.index < o_N.index$.

$\square$

Theorem 3 only modifies the Condition 2b of Lemma 2. Therefore, Lemma 2 along with Theorem 3 imposes restrictions on every Xmodule M which are only slightly weaker than the restrictions required by Definition 1. Condition 1 in Lemma 2 corresponds to Rule 1 of Definition 1. Conditions 2a and 2b are the cases permitted by Rule 2. Condition 2c, however, corresponds to the special case of callbacks or calling a method from the same Xmodule, which is not permitted by Definition 1. This case is modeled differently, as we explained in Section 2.

The *OAT* type system is a best-effort type system to check for the restrictions required by Definition 1. The *OAT* type system can not fully guarantee, however, that a type-checked program does not violate Definition 1. Specifically, the *OAT* type system can not always detect the following violations statically. First, if the program does not have unique owner indices, then L may instantiate both M and N with the same index. Then, by Lemma 2, M and N, can call each other's methods, and we can get cyclic dependencies between Xmodules.[6] Second, the program may perform improper callbacks. Say a method from M calls back to method B from L. An improper callback B can call a method of N, even though the type system knows that M is to the right of N. In both cases, the type system allows a program with cyclic dependency between Xmodules to pass the type checks, which is not allowed by Definition 1.

[5] Note that L could be the `world` Xmodule, in which case both o_N and o_Q were created in the `main` function using the `world` keyword.

[6] Since all non-Xmodule objects are implicitly assigned higher indices than their Xmodule siblings, these non-Xmodule objects can not introduce cyclic dependencies between Xmodules.

To have an ownership-aware TM which guarantees exactly Definition 1, one needs to impose additional dynamic checks. The runtime system can use the ownership tags to build a module tree during runtime, and use this module tree to perform dynamic checks to verify that every Xmodule has unique owner indices and contains only proper callbacks. The runtime system can do this by dynamically inferring indices according to which Xmodule calls which other Xmodule, and reporting an error if there is any circular calling.[7]

4. COMPUTATIONS WITH Xmodules

In this section, we formally define the structure of transactional programs with Xmodules. This section converts the informal explanation from Section 2 into a formal model that we later use to prove properties of ownership-aware TM. First, we briefly review the transactional computation framework described in [2]. We then add Xmodules and ownership to this framework, and provide the formal statement of Definition 1.

Transactional Computations

In the framework from [2], the execution of a program is modeled using a "computation tree" $\mathcal{C}$ that summarizes the information about both the control structure of a program and the nesting structure of transactions, and an "observer function" Φ 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 ***read predicate*** $R(v,\ell)$ if v reads from location ℓ, while v satisfies the ***write predicate*** $W(v,\ell)$ if v writes to ℓ. 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 ℓ, we abstract away the values by using an ***observer function*** Φ. For a memory operation v that accesses a memory location ℓ, the node $\Phi(v)$ is defined to be the operation that wrote the value of ℓ that v sees.

We define several structural notations on the computation tree $\mathcal{C}$. Denote the ***root*** of $\mathcal{C}$ as $\texttt{root}(\mathcal{C})$. For any tree node X, let $\texttt{ances}(X)$ denote the set of all X's ancestors (including X itself) in $\mathcal{C}$. Similarly, let $\texttt{desc}(X)$ denote the set of all X's descendants, including X itself. Denote the set of proper ancestors of X by $\texttt{pAnces}(X)$. For any tree node X, 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 $\texttt{xAnces}(X) = \texttt{ances}(X) \cap \texttt{xactions}(\mathcal{C})$. Denote the ***least common ancestor*** of two nodes $X_1, X_2 \in C$ by $\texttt{LCA}(X_1, X_2)$. 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 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 3 show a computation tree and its corresponding computation dag.

Classical theories on serializability refer to a particular execution order for a program as a ***history*** [12]. 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 a different topological sort $\mathcal{S}'$ of $G(\mathcal{C})$ which has all transactions appearing contiguously, but which is still "consistent" with the observer function associated with $\mathcal{S}$.

[7] It is possible to statically check for unique owner indices by imposing additional restrictions on the program. We opted, however, to describe a more flexible programming model with weaker static guarantees.

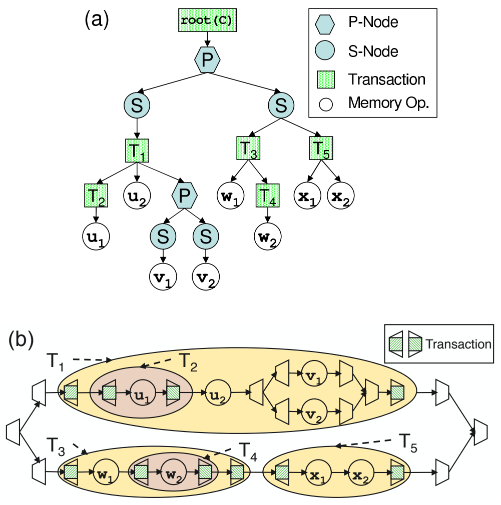

Figure 3. A sample (a) computation tree $\mathcal{C}$ and (b) its corresponding dag $G(\mathcal{C})$.

Xmodules and Computation Tree

As mentioned in Section 2, in this paper, we consider programs that contain Xmodules. In our theoretical framework, we consider traces generated by a program which is organized into a set $\mathcal{N}$ of Xmodules. Each Xmodule $M \in \mathcal{N}$ has some number of methods and a set of memory locations associated with it.

We partition the set of all memory locations $\mathcal{L}$ into sets of memory owned by each Xmodule. Let $\texttt{modMemory}(M) \subseteq \mathcal{L}$ denote the set of memory locations owned by M. For a location $\ell \in \texttt{modMemory}(M)$, we say that $\texttt{owner}(\ell) = M$. When a method of Xmodule M is called by a method from a different Xmodule, a safe-nested transaction T is generated.[8] We use the notation $\texttt{xMod}(T) = M$ to associate the instance T with the Xmodule M. We also define the instances associated with M as

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

As mentioned in Section 2, Xmodules of a program are arranged as the module tree, denoted by $\mathcal{D}$. Each Xmodule is assigned an $\texttt{xid}$ according to a left-to-right depth-first tree walk, with the root of $\mathcal{D}$ being $\texttt{world}$ with $\texttt{xid} = 0$. Denote the parent of Xmodule M in $\mathcal{D}$ as $\texttt{modParent}(M)$, and the ancestors of M as $\texttt{modAnces}(M)$ (include M itself). Similarly, let $\texttt{modDesc}(M)$ be the set of M's descendants. We say that $\texttt{xMod}(\texttt{root}(\mathcal{C})) = \texttt{world}$, i.e., the root of the computation tree is a transaction associated with the $\texttt{world}$ Xmodule.

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 2.

DEFINITION 2 (Formal Restatement of Definition 1). *A program with a module tree $\mathcal{D}$ should generate only traces $(\mathcal{C}, \Phi)$ which satisfy the following rules:*

1. *For any memory operation v which accesses a memory location ℓ, 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 $\texttt{xMod}(X) = M$ and $\texttt{xMod}(Y) = N$. We can have $X = \texttt{xparent}(Y)$ only if $\texttt{modParent}(N) \in \texttt{modAnces}(M)$, and $\texttt{xid}(M) < \texttt{xid}(N)$.*

5. THE OAT MODEL

In this section, we describe 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

[8] As we explained in Section 2, callbacks are handled differently.

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, although these semantics can be used to guide an 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 node to 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.

5.1 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 4 (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 4 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.

At any time t, each internal node $A \in \texttt{spNodes}(t,\mathcal{C})$ has a ***status*** field $\texttt{status}[A]$. These status fields change with time. 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 5.2. 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 ***read set*** $\texttt{R}(t,T)$ and a ***write set*** $\texttt{W}(t,T)$ for T. The read set $\texttt{R}(t,T)$ is a set of pairs (ℓ,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 ℓ. We define $\texttt{W}(t,T)$ similarly. We represent main memory as the read set/write set 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 $\ell \in \mathcal{L}$.

In addition to the basic read and write sets, we also define ***module read set*** and ***module write set*** for all transactions $T \in \texttt{activeX}(t,\mathcal{C})$. Module read set is defined 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 write set*** as

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

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 (ℓ,v) for any location ℓ. 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

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

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

5.2 Constructing the Computation Tree

In the *OAT* model, the runtime constructs the computation tree in a straightforward fashion as instructions are issued. For completeness, however, we give a detailed description of this construction.

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

A new internal node is created if the *OAT* model picks ready node X and X issues 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] = $ 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] = $ 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 $\texttt{ready}(\mathcal{C})$, changes the status of both Z and $\texttt{parent}(Z)$ to SYNCHED, and adds $\texttt{parent}(\texttt{parent}(Z))$ to $\texttt{ready}(\mathcal{C})$.

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] = $ PENDING, then X can issue an xend to change $\texttt{status}[X]$ to COMMITTED. Otherwise, $\texttt{status}[X] = $ 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 read sets and write sets, which we describe later in Section 5.4.

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.

This construction of the tree guarantees a few properties.

First, the sequence of instructions $\mathcal{S}$ generated by the *OAT* model is a valid topological sort of the computation dag $G(\mathcal{C})$. Second, the *OAT* model generates a tree of a canonical form, where the root node of the tree is a transaction, all transactions are S-nodes and every P-node has exactly two nontransactional S-node children. This canonical form is imposed for convenience of description; it is not important for any theoretical results. Finally, the *OAT* model maintains the invariant the active nodes form a tree, with the ready nodes at the leaves. This property is important for the correctness of the *OAT* model.

5.3 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 a potential memory operation 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 5.5. Otherwise, v does not generate a conflict and observes the value ℓ from $\texttt{R}(Y)$, where Y is the closest ancestor of v with ℓ 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 (ℓ,v) to both $\texttt{R}(X)$ and $\texttt{W}(X)$.

5.4 Ownership-Aware Transaction Commit

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

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.

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

Unique Committer Property

Definition 2 guarantees certain properties of the computation tree which are essential to the ownership-aware commit mechanism. Theorem 5 proves that every memory operation has one and only one transaction that is responsible for committing the memory operation. The proof of the theorem requires the following lemma.

LEMMA 4. *Given a computation tree $\mathcal{C}$, for any $T \in \texttt{xactions}(\mathcal{C})$, let $S_T = \{\texttt{xMod}(T') \, : \, T' \in \texttt{xAnces}(T)\}$. Then $\texttt{modAnces}(\texttt{xMod}(T)) \subseteq S_T$.*

PROOF. We prove this fact by induction on the nesting depth of transactions T 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. Then, by Rule 2 of Definition 2, we have $\texttt{modParent}(\texttt{xMod}(T^*)) \in \texttt{modAnces}(\texttt{xMod}(T))$. 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, using the inductive hypothesis, we have $\texttt{modAnces}(\texttt{xMod}(T^*)) \subseteq S_{T^*}$. $\square$

THEOREM 5. *If a memory operation v accesses a memory location ℓ, then there exists a unique transaction $T^* \in \texttt{xAnces}(v)$, 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 ℓ.

*This transaction T^* is the **committer** of memory operation v, denoted $\texttt{committer}(v)$.*

PROOF. This result follows from the properties of the module tree and computation tree stated in Definition 2.

Let $T = \texttt{xparent}(v)$. First, by Definition 2, Rule 1, we know $\texttt{owner}(\ell) \in \texttt{modAnces}(\texttt{xMod}(T))$. We know $\texttt{modAnces}(\texttt{xMod}(T)) \subseteq S_T$ by Lemma 4. Thus, 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, i.e., 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 $\texttt{xid}(\texttt{xMod}(X_i)) < \texttt{xid}(\texttt{xMod}(X_{i-1}))$, that is, the xids strictly decrease walking up the tree from T. Thus, there can only be one ancestor transaction T^* of T with $\texttt{xid}(\texttt{xMod}(T^*)) = \texttt{xid}(\texttt{owner}(\ell))$.

To check the second condition of Theorem 5, consider any $X \in \texttt{pAnces}(T^*) \cap \texttt{xactions}(\mathcal{C})$. By Rule 1, X can access ℓ directly only if $\texttt{owner}(\ell) \in \texttt{modAnces}(\texttt{xMod}(X))$ implying that $\texttt{xid}(\texttt{owner}(\ell)) \leq \texttt{xid}(\texttt{xMod}(X))$. But we know that $\texttt{owner}(\ell) = \texttt{xMod}(T^*)$ and $\texttt{xid}(\texttt{xMod}(T^*)) > \texttt{xid}(\texttt{xMod}(X))$. $\square$

Intuitively, $T^* = \texttt{committer}(v)$ is the transaction which "belongs" to the same Xmodule as the location ℓ which v accesses, and is "responsible" for committing v to memory and making it visible to the world. The second condition of Theorem 5 states that no ancestor transaction of T^* in the call stack can ever directly access ℓ; thus, it is "safe" for T^* to commit ℓ.

5.5 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 X might not abort immediately; instead, it might continue to issue more instructions after its status has changed to PENDING_ABORT. Later, it will be useful to refer to the set of operations a transaction X issues while its status is PENDING_ABORT.

DEFINITION 4. *The set of operations issued by X or descendants of X after $\texttt{status}[X]$ changes to PENDING_ABORT are called X's **abort actions**. This set is denoted by $\texttt{abortactions}(X)$.*

The PENDING_ABORT status allows X to compensate for the safe-nested transactions that may have committed; if transaction Y is nested inside X, then the abort actions of X contain the compensating action of Y. Eventually a PENDING_ABORT transaction issues an xend instruction, which changes its status from PENDING_ABORT to ABORTED.

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 $\texttt{xparent}(v)$, or T_u. In the latter case, v 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.

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 7.

6. SERIALIZABILITY BY MODULES

In this section, we define ***serializability by modules***, a definition inspired by the database notion of multilevel serializability (e.g., as described in [13]). First, we describe the definition of serializability in the transactional computation framework, as given in [2]. 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.

6.1 Transactional Computations and Serializability

In [2], 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 Φ", and all transactions appear contiguous in the order $\mathcal{S}$. In this section, we give more precise, formal definitions of this concept.

Content Sets

We first describe some notation needed to formally describe serializability by modules. All definitions in this section are *a posteriori*, i.e., they are defined on the computation tree after the program has finished executing.

We define "content" sets for every transaction T by partitioning $\texttt{memOps}(T)$ (all the memory operations enclosed inside T including those belonging to its nested transactions) into three sets: $\texttt{cContent}(T)$, $\texttt{oContent}(T)$ and $\texttt{aContent}(T)$. For any $u \in \texttt{memOps}(T)$, we define the content sets based on the final status of transactions in $\mathcal{C}$ that one visits when walking up the tree from u to T.

DEFINITION 5. *For any transaction T and memory operation u, define the sets* $\texttt{cContent}(T)$, $\texttt{oContent}(T)$, *and* $\texttt{aContent}(T)$ *according the* $\texttt{ContentType}(u,T)$ *procedure:*

```
  ContentType$(u,T)$        $\triangleright$ For any $u \in \texttt{memOps}(T)$
1 $X \leftarrow \texttt{xparent}(u)$
2 while $(X \neq T)$
3     if $(X$ is ABORTED$)$        return $u \in \texttt{aContent}(T)$
4     if $(X = \texttt{committer}(u))$  return $u \in \texttt{oContent}(T)$
5     $X \leftarrow \texttt{xparent}(X)$
6 return $u \in \texttt{cContent}(T)$
```

Recall that in the *OAT* model, the safe-nested commit of T commits some memory operations in an open-nested fashion, to $\texttt{root}(\mathcal{C})$, and some operations in a closed-nested fashion, to $\texttt{xparent}(T)$. Informally, $\texttt{oContent}(T)$ 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's 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., are ***hidden*** from) memory operations outside of T.

DEFINITION 6. *For $u \in \texttt{memOps}(\mathcal{C}), v \in V(\mathcal{C})$, let $X = \texttt{xLCA}(u,v)$. We say that u is **hidden** from v 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 7. *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 8. *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** Φ.*

In other words, the transactional last writer of a memory operation u which accesses location ℓ, is the last write v to location ℓ in the order $\mathcal{S}$, except we skip over writes w which are hidden from (i.e., aborted with respect to) u. Intuitively, Definition 8 requires that there exists an order $\mathcal{S}$ explaining all the memory operations of the computation.

Serializability

DEFINITION 9. *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 Φ), 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.

6.2 Defining Serializability by Modules

In [2], 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 Φ, 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. This definition of serializability is the "correct definition" for flat or closed-nested transactions. This definition of serializability is too strong, however, for ownership-aware transactions. A TM system that enforces this definition of serializability can not ignore lower-level memory accesses when detecting conflicts for higher-level transactions.

Instead, we describe a definition of serializability by modules which checks for correctness of one Xmodule at a time. Given a trace $(\mathcal{C},\Phi)$, for each Xmodule M, we transform the tree $\mathcal{C}$ into a new tree $\texttt{mTree}(\mathcal{C},M)$. The tree $\texttt{mTree}(\mathcal{C},M)$ is constructed in such a way as to ignore memory operations of Xmodules which are lower-level than M, and also to ignore all operations which are hidden from transactions of M. For each Xmodule M, we check that the transactions of M in the trace $(\texttt{mTree}(\mathcal{C},M),\Phi)$ is serializable. If the check holds for all Xmodules, then trace $(\mathcal{C},\Phi)$ is said to be serializable by modules.

Definition 10 formalizes the construction of $\texttt{mTree}(\mathcal{C},M)$.

DEFINITION 10. *For any computation tree $\mathcal{C}$, let $\texttt{mTree}(\mathcal{C},M)$ be the result of modifying $\mathcal{C}$ as follows:*

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

The intuition behind Condition 1 of Definition 10 is the following. When looking at Xmodule M, we throw away memory operations belonging to a lower-level Xmodule N, since by Theorem 5, transactions of M can never directly access the same memory as those operations anyway. In Condition 2, we ignore the content of any aborted transactions nested inside transactions of M; 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 M.

Lemma 6 argues that if a trace $(\mathcal{C},\Phi)$ is sequentially consistent, then $(\texttt{mTree}(\mathcal{C},M),\Phi)$ is a valid trace; an operation u that remains in the trace never attempts to observe a value from a $\Phi(u)$ which was turned into a nop due to Definition 10. In addition, the transformed trace is also sequentially consistent.

LEMMA 6. *Let $(\mathcal{C},\Phi)$ be any sequentially consistent trace. Then for any Xmodule M, $(\texttt{mTree}(\mathcal{C},M),\Phi)$ is a valid trace. In other words, if $u \in \texttt{memOps}(\texttt{mTree}(\mathcal{C},M))$, then $\Phi(u) \in \texttt{memOps}(\texttt{mTree}(\mathcal{C},M))$. Furthermore, any $\mathcal{S}$ which is sequentially consistent with respect to Φ in $(\mathcal{C},\Phi)$ is also sequentially consistent with respect to Φ in $(\texttt{mTree}(\mathcal{C},M),\Phi)$.*

PROOF. In the new tree $\texttt{mTree}(\mathcal{C},M)$, pick any $u \in \texttt{memOps}(\texttt{mTree}(\mathcal{C},M))$ 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 of Definition 10, then we know because v accessed a memory location ℓ where $\texttt{xid}(\texttt{owner}(\ell)) > \texttt{xid}(M)$. Since u must access the same location ℓ, u must also be converted into a nop.

If v was turned into a nop in Step 2 of Definition 10, then $v \in \texttt{aContent}(T)$ for some $\texttt{xMod}(T) = M$. 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 $0 \leq j \leq k$ must have $\texttt{status}[Y_j] = \texttt{ABORTED}$. Since T is a proper ancestor of X, $X = Y_x$ for some x satisfying $0 \leq x \leq k$.
   (a) If $\texttt{status}[Y_j] = \texttt{ABORTED}$ for any j satisfying $0 \leq j < x$, 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 7, we know $\neg vHu$, leading to a contradiction.
   (b) If Y_j is $\texttt{ABORTED}$ for any j satisfying $x \leq j \leq k$, 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 Φ is the transactional last writer according to $\mathcal{S}$ for $(\mathcal{C},\Phi)$, it is still the transactional last writer for $(\texttt{mTree}(\mathcal{C},M),\Phi)$ because the memory operations which are not turned into nops remain in the same relative order. Thus, the last condition is satisfied. $\square$

Note that Lemma 6 *depends on* the restrictions on Xmodules described in Definition 2. Without this structure of modules and ownership, the construction of Definition 10 is not guaranteed to generate a valid trace.

Finally, we can define serializability by modules.

DEFINITION 11. *A trace $(\mathcal{C},\Phi)$ is **serializable by modules** if it is sequentially consistent, and if for all Xmodules M in $\mathcal{D}$, there exists a topological sort $\mathcal{S}$ of $\mathcal{C}_M = \texttt{mTree}(\mathcal{C},M)$ such that:*

1. *$\mathcal{S}$ is sequentially consistent with respect to Φ, and*
2. *For the tree $\mathcal{C}_M$, $\forall T \in \texttt{modXactions}(M)$ and $\forall v \in V(\mathcal{C}_M)$, 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 M, there exists a sequentially consistent order $\mathcal{S}$ for the trace $(\texttt{mTree}(\mathcal{C},M),\Phi)$ such that all transactions of M are contiguous in $\mathcal{S}$.

6.3 *OAT* Model Guarantees Serializability by Modules

In this section, we show that the *OAT* model described in Section 5 generates traces $(\mathcal{C},\Phi)$ that are serializable by modules, i.e., that satisfy Definition 11. The proof of this fact consists of three steps. First, we generalize the notion of "prefix race-freedom" described in [2], 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 prefix races. These definitions are essentially the same as those in [2], except adapted for a system with an ownership-aware commit mechanism instead of an open-nested commit mechanism.

DEFINITION 12. *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 ℓ, and one of v,w writes to ℓ.*

DEFINITION 13. *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 Φ), 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 7. *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 Φ.*

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. Informally, Lemma 8 states that, if a memory operation u that reads (writes) location ℓ is in the cContent(T) for some transaction T, then ℓ belongs to the read set (write set) of some active transaction under T's subtree between the time when the memory operation is performed and the time when T ends.

LEMMA 8. *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 ℓ at step t_0. Let t_f be step when $\texttt{xend}(T)$ or $\texttt{xabort}(T)$ happens. At any time t such that $t_0 \le t < t_f$ there exists some $T' \in \texttt{xDesc}(T) \cap \texttt{activeX}(t,\mathcal{C})$ (i.e., T' is an active transactional 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 that $u \in \texttt{cContent}(T)$ and since T completes at time t_f, for every j such that $1 \le j < k$, there exists a unique time t_j (satisfying $t_0 \le t_j < t_f$) when an xend changes $\texttt{status}[X_j]$ from PENDING to COMMITTED; otherwise, we would have $u \in \texttt{aContent}(T)$.

Also, by Theorem 5 and Definition 5, we know $\texttt{committer}(u) \in \texttt{xAnces}(T)$, i.e., none of the X_j's will commit location ℓ 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, (ℓ,u) is added to $\texttt{R}(X_1)$. In general, at time t_j, the ownership-aware commit mechanism, as described in Section 5.4, will propagate ℓ 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 8, $T' = X_j$. 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 $\ell \in \texttt{W}(t,T')$. □

Angelina: Check this.

We use Theorem 7 and Lemma 8 to prove that the *OAT* model generates traces which are prefix race-free.

THEOREM 9. *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 13, we know by Theorem 7 that the *OAT* model generates an order $\mathcal{S}$ which is sequentially consistent with respect to Φ.

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 12, 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 ℓ, 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.

There are three cases for v and w. First suppose $W(v,\ell)$ and $R(w,\ell)$. Since $t_w < t_v < t_{endT}$, by Lemma 8, at time t_v, ℓ 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 10. *Any trace $(\mathcal{C},\Phi)$ which is prefix race-free is also serializable by modules.*

PROOF.

First, by Definition 10 and Lemma 6, 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 $(\texttt{mTree}(\mathcal{C},M),\Phi)$ for any Xmodule M. Now we shall argue that for any Xmodule M, we can transform $\mathcal{S}$ into $\mathcal{S}_M$ such that all transactions in $\texttt{xactions}(M)$ appear contiguous in $\mathcal{S}_M$.

Consider a prefix-race free sort $\mathcal{S}$ of $(\texttt{mTree}(\mathcal{C},M),\Phi)$ which has k nodes v which violate the second condition of Definition 11. We show how to construct a new order $\mathcal{S}'$ which is still a prefix race-free sort of $(\texttt{mTree}(\mathcal{C},M),\Phi)$, but which has only $k-1$ violations.

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

1. Of all transactions $T \in \texttt{modXactions}(M)$ 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 $(\texttt{mTree}(\mathcal{C},M),\Phi)$, 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 Φ (i.e., that Φ 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 ℓ. 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 ℓ unless both w and v read from ℓ. 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 $\texttt{mTree}(\mathcal{C},M)$, 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 10 will turn w into a nop). Therefore, by Definition 12, unless w and v both read from ℓ, 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 11 one at a time, we can construct a sort $\mathcal{S}_M$ which satisfies serializability by modules by eliminating all violations. $\square$

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

THEOREM 11. *Any trace $(\mathcal{C},\Phi)$ generated by the OAT model is serializable by modules.*

PROOF. By Theorem 9, the *OAT* model generates only trace $(\mathcal{C}, \Phi)$ which are prefix race-free. By Theorem 6.3, any trace $(\mathcal{C}, \Phi)$ which is prefix race-free is serializable by modules. $\square$

6.4 Abstract Serializability

By Theorem 11, the *OAT* model guarantees serializability by modules. We now relate this definition to the notion of *abstract serializability* used in multilevel database systems [13]. As we mentioned in Section 1, the ownership-aware commit mechanism is 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 [13] assumes that the program is divided into levels, and that a transaction at level i can only call a transaction at level $i+1$.[9] 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. The transactions at the lowest level (say k) are naturally serializable; call this schedule $\mathcal{Z}_k$. Given a serializable schedule $\mathcal{Z}_{i+1}$ of level-$i+1$ transactions, the schedule is said to be serializable at level i if all transactions in $\mathcal{Z}_{i+1}$ can be reordered, obeying all commutativity rules, to obtain a serializable order $\mathcal{Z}_i$ for level-i transactions. The original schedule is said to be abstractly serializable if it is serializable for all levels.

These commutativity rules might be specified using abstract locks [11]: if two transactions can not commute, then they grab the same abstract lock in a conflicting manner. In the application described in Section 2, for instance, transactions calling `insert` and `remove` on the `BST` using the same key do not commute and should grab the same write lock. Although abstract locks are not explicitly modeled in the *OAT* model, we can model transactions acquiring the same abstract lock as transactions writing to a common memory location ℓ.[10] Locks associated with an Xmodule M are owned by $\texttt{modParent}(M)$. A module M is said to be ***properly locked*** if the following is true for all transactions T_1, T_2 with $\texttt{xMod}(T_1) = \texttt{xMod}(T_2) = M$: if T_1 and T_2 do not commute, then they access some $\ell \in \texttt{modMemory}(\texttt{modParent}(M))$ in a conflicting manner.

If all transactions are properly locked, then serializability by modules implies abstract serializability (as defined above) in the special case when the module tree is a chain (i.e., each non-leaf module has exactly one child). Let $\mathcal{S}_i$ be the sort $\mathcal{S}$ in Definition 11 for Xmodule M with $\texttt{xid}(M) = i$. This $\mathcal{S}_i$ corresponds to $\mathcal{Z}_i$ in the definition of abstract serializability.

In the general case for ownership-aware TM, however, by Rule 2 of Definition 1, we know a transaction at level i might call transactions from multiple levels $x > i$, not just $x = i+1$. Thus, we must change the definition of abstract serializability slightly; instead of reordering just $\mathcal{Z}_{i+1}$ while serializing transactions at level-i, we have to potentially reorder $\mathcal{Z}_x$ for all x where transactions at level i can call transactions at level x. Even in this case, if every module is properly locked (by the same definition as above), one can show serializability by modules guarantees abstract serializability.

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

[9] We assume level number increases as you go from a higher level to a lower-level to be consistent with our numbering of `xid`. In the literature (e.g. [13]), levels typically go in the opposite direction.

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

7. DEADLOCK FREEDOM

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

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; a scheduler in the TM runtime can 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 (i.e., when the status of T_2 becomes 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. Thus T_1 and T_2 can both wait on each other, and 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 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 $\mathcal{F}$ 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 $\mathcal{F}$ can not complete the program no matter what it does. To eliminate programs which have infinite loops, we only consider ***bounded programs***.

DEFINITION 14. *We say that a program is* ***bounded*** *for an operational model $\mathcal{F}$ if any computation tree that $\mathcal{F}$ 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 or COMMITTED.*

Even if the program is bounded, it might run forever if it ***livelocks***. We use the notion of a ***schedule*** to distinguish livelocks from semantic deadlocks.

DEFINITION 15. *A* ***schedule*** *Γ on some time interval $[t_0, t_1]$ is the sequence of nondeterministic choices made by an operational model in the interval.*

An operational model $\mathcal{F}$ makes two types of nondeterministic choices. First, at any time t, $\mathcal{F}$ 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, $\mathcal{F}$ nondeterministically chooses to abort either $\texttt{xparent}(u)$ or T. This nondeterministic choice models the contention manager of the TM runtime. A program may livelock if $\mathcal{F}$ repeatedly makes "bad" scheduling choices.

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.

DEFINITION 16. *Consider an $\mathcal{F}$ executing a bounded computation. We say that $\mathcal{F}$ does not exhibit a* ***semantic deadlock*** *if for all finite sequences of t_0 instructions that $\mathcal{F}$ can issue that generates some intermediate computation tree $\mathcal{C}_0$, there exists a finite schedule Γ on $[t_0, t_1]$ such that $\mathcal{F}$ 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, $\mathcal{F}$ can execute each transaction serially and complete the computation.

Restrictions to Avoid Semantic Deadlock

The general *OAT* model described in Section 5 exhibits semantic deadlock because it may enter a state where two parallel aborting transactions T_1 and T_2 keep each other from completing their aborts. For a restricted set of programs, where a PENDING_ABORT transaction T never accesses new memory belonging to Xmodules at $\texttt{xMod}(T)$'s level or higher, however, 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 17. *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 ℓ and $\texttt{owner}(\ell) \in \texttt{modAnces}(\texttt{xMod}(T))$, then (1) if v is a read, then $\ell \in \texttt{R}(T)$, and (2) if v is a write then $\ell \in \texttt{W}(T)$.*

Intuitively, Definition 17 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 ℓ which is owned by any Xmodules which are ancestors of $\texttt{xMod}(T)$ (including $\texttt{xMod}(T)$ itself), unless ℓ is already in the read (or write set) of T.

First, we show that the properties of Xmodules from Theorem 5 in combination with the ownership-aware commit mechanism imply that transaction read sets and write sets exhibit nice properties. In particular, we have Corollary 12, which states that a location ℓ can appear in the read set of a transaction T only if T's Xmodule is a descendant of $\texttt{owner}(\ell)$ in the module tree $\mathcal{D}$.

COROLLARY 12. *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 5, and induction on how a location ℓ can propagate into readsets and writsets using the ownership-aware commit mechanism. $\square$

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

LEMMA 13. *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 $\texttt{xid}(\texttt{xMod}(T')) > \texttt{xid}(\texttt{xMod}(T))$.*

PROOF. Suppose $v \in \texttt{abortactions}(T)$ accesses a memory location ℓ with $\texttt{owner}(\ell) = M$. Since $\texttt{abortactions}(T) \subseteq \texttt{memOps}(T)$, by the properties of Xmodules given in Definition 2, we know that either $M \in \texttt{modAnces}(\texttt{xMod}(T))$, or $\texttt{xid}(M) > \texttt{xid}(\texttt{xMod}(T))$. If $M \in \texttt{modAnces}(\texttt{xMod}(T))$, then by Definition 17, T already had ℓ 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 $\texttt{xid}(M) > \texttt{xid}(\texttt{xMod}(T))$. If v conflicts with some other transaction T', then T' has ℓ in its read or write set. Therefore, from Corollary 12, $\texttt{xMod}(T') \in \texttt{modDesc}(M)$. Thus, we have $\texttt{xid}(\texttt{xMod}(T')) > \texttt{xid}(M) > \texttt{xid}(\texttt{xMod}(T))$. $\square$

THEOREM 14. *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 Γ 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, $\texttt{status}[T] = $ PENDING_ABORT for all active transactions T. Otherwise, the first phase of the schedule Γ is to make this status change for all active transactions T.

For a module tree $\mathcal{D}$ with $k = |\mathcal{D}|$ Xmodules (including the `world`), we construct a schedule Γ with k phases, numbered $k-1, k-2, \ldots 1, 0$. 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 $\texttt{xid}(\texttt{xMod}(T)) > i$, i.e., no instances T from Xmodules with `xid` larger than i. During phase i, we finish aborting all active transaction instances T with $\texttt{xid}(\texttt{xMod}(T)) = i$. By Lemma 13, any abort action for a T, where $\texttt{xid}(\texttt{xMod}(T)) = i$, can only conflict with a transaction instance T' from a lower-level Xmodule, where $\texttt{xid}(\texttt{xMod}(T')) > i$. Since the schedule Γ executes serially, and since by the inductive hypothesis we have already finished all active transaction instances from lower levels, phase i can finish without generating any conflicts. $\square$

Restrictions on compensating actions

If transactions $Y_1, Y_2, \ldots Y_j$ are nested inside transaction X and X aborts, typically abort actions of X simply consists of compensating actions for $Y_1, Y_2, \ldots Y_j$. Thus, restrictions on abort actions translate in a straightforward manner to restrictions on compensating actions: a compensating action for a transaction Y_i (which is part of the abort action of X), should not read (write) any memory owned by $\texttt{xMod}(X)$ or its ancestor Xmodules 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.

8. CONCLUSIONS

In this paper, we describe ownership-aware transactions, which provide a disciplined methodology for open nesting while guaranteeing abstract serializability. In this section, we describe two other approaches for improving open-nested transactions, and distinguish them from ownership-aware transactions.

In [11], Ni, et al. propose using an `open_atomic` class to specify open-nested transactions in a Java-like language with transactions. Since the private fields of an object with an `open_atomic` class type can not be directly accessed outside of that class, one can think of the `open_atomic` class as defining an Xmodule. This mapping is not exact, however, because neither the language or TM system restrict exactly what memory can be passed into a method of an `open_atomic` class, and the TM system performs a vanilla open-nested commit for a nested transaction, not a safe-nested commit. Thus, it is unclear what exact guarantees are provided with respect to serializability and/or deadlock freedom.

Herlihy and Koskinen in [5] describe a technique of transactional boosting which allows transactions to call methods from a nontransactional module M. Roughly, as long as M is linearizable and its methods have well-defined inverses, the authors show that the execution appears to be "abstractly serializable." Boosting does not, however, address the cases when the lower-level module M writes to memory owned by the enclosing higher-level module, or when programs have more than two levels of modules.

Acknowledgements

We thank James Noble of Victoria University of Wellington, Derek Rayside, Martin Rinard, Amy Williams, and Charles Leiserson and other members of the Supercomputing Technologies Group at MIT CSAIL for helpful discussions and comments on the paper. We also thank all the reviewers of this and prior versions of the paper for their comments. In particular, we are grateful to Bill Scherer for his help in improving the paper.

REFERENCES

[1] K. Agrawal, I.-T. A. Lee, and J. Sukha. Safe open-nested transactions through ownership. Technical Report MIT-CSAIL-TR-2008-038, Laboratory of Computer Science and Artificial Intelligence, Massachusetts Institute of Technology, June 2008. Available online at
`http://supertech.csail.mit.edu/~angelee/safeTech.pdf`.

[2] 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.

[3] 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.

[4] 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.

[5] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 207–216, New York, NY, USA, Feb 2008. ACM.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

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

[13] 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 7: if the *OAT* model generates a trace $(\mathcal{C}, \Phi)$ and a topological sort order $\mathcal{S}$, then $\mathcal{S}$ satisfies Definition 8, i.e., $\mathcal{S}$ is sequentially consistent with respect to Φ.

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, read set, and write sets. Finally, we use these invariants to prove Theorem 7.

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 ℓ in their read set. 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 5 and define a set of dynamic content sets.

DEFINITION 18. *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 $u \in$ memOps($t,T$)
1 $X \leftarrow$ xparent($u$)
2 while ($X \neq T$)
3     if $X \in$ activeX($t,\mathcal{C}$),         return $u \in$ vContent($t,T$)
4     if $X \in$ aborted($t,\mathcal{C}$),         return $u \in$ aContent($t,T$)
5     if ($X =$ committer($u$))      return $u \in$ oContent($t,T$)
6     $X \leftarrow$ xparent($X$)
7 return $u \in$ cContent($t,T$)
```

The difference between Definition 18 and the previous statement in Definition 5 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 $t \geq t^*$.

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 ℓ.

THEOREM 15. *At all times t, for all memory locations $\ell \in \mathcal{L}$, 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 read sets or write sets. An `xbegin` preserves the invariants because it creates new transactions T with empty read sets and write sets. The `xabort` instruction preserves the invariants because it can only remove transactions from $\texttt{readers}(t,\ell)$ or $\texttt{W}(t,\ell)$. □

The following invariant shows that, informally, the read sets of transactions act as caches for pairs (ℓ,u) stored in write sets.

LEMMA 16. *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 16 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 write set, 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 the property that the ready nodes are the leaves of tree of active nodes. Thus, the `xabort` must preserve the invariant.

A successful `read` operation v observes the value from the closest transactional ancestor X which has location ℓ in its read set. The only transaction whose read set changes is $\texttt{xparent}(v)$. The invariant is preserved because $\texttt{xAnces}(\texttt{xparent}(v)) \supseteq \texttt{xAnces}(X)$, and since the `read` does not change any write sets.

A successful `write` operation v only changes the write set of $\texttt{xparent}(v)$; this `write` can not break the invariant without generating a conflict.

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 which has its read set or write set change after the `xend` (i.e., 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 $Y = \texttt{xparent}(X)$. The `xend` merges

X's read and write sets into Y's read and write sets, respectively; using Theorem 15, it is straightforward to show that the invariant is preserved for Y.

For all other transactions $T \in \texttt{readers}(t, \ell)$ with $T \neq Y$, since the read set or write set 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 read set or write set can change is $Y = \texttt{root}(\mathcal{C})$. But the only way to break the invariant is if X commits a pair (ℓ, v) from $\texttt{W}(t-1, X)$ to $\texttt{root}(\mathcal{C})$, which corrupts the version $(\ell, u) \in \texttt{R}(t-1, T)$, for some transaction T parallel to X. But then, we would violate Theorem 15, and should have had a conflict earlier.

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

Theorem 17 characterizes when a transaction should have a location in its write set.

THEOREM 17. *At any time t, consider any transaction $T \in \texttt{activeX}(t, \mathcal{C})$ and any memory location ℓ such that $\texttt{xid}(\texttt{owner}(\ell)) \leq \texttt{xMod}(T)$. Let $S_\ell(t) = \{u \in \texttt{memOps}(t, \mathcal{C}) : W(u, \ell)\}$. Exactly one of the following cases holds:*

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

PROOF.

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

Note that because we assume $\texttt{xid}(\texttt{owner}(\ell)) \leq \texttt{xMod}(T)$, $S_\ell(t) \cap \texttt{oContent}(t, T) = \emptyset$, i.e., the theorem is only concerned with memory locations ℓ which belong to T's open content. Because of the properties of ownership and Xmodules, any location ℓ with $\texttt{xid}(\texttt{owner}(\ell)) > \texttt{xMod}(T)$ can never propagate into T's write set anyway. □

The intution for Theorem 17 lies mostly in Case 2; if at time t a pair (ℓ, u) is the write set of a transaction T, then u is the last write to ℓ in T's subtree which is "committed with respect to" T. Any v which writes to ℓ after t_u (the time u occurs) must belong to T's subtree; otherwise, there would have been a conflict. Furthermore, any v which happens after t_u must still be aborted or pending with respect to T (i.e., $v \in \texttt{aContent}(t, T) \cup \texttt{vContent}(t, T)$); otherwise, v should replace u in T's write set.

Case 3 says the write set of T does not contain a location ℓ if no memory operation in T's subtree commits ℓ to T. Case 1 of Theorem 17 handles the special case of the root.

A.3 Proof of Sequential Consistency

Finally, we can use the invariants from Lemma 16 and Theorem 17 to prove Theorem 7.

PROOF. [Theorem 7]

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 $R(v, \ell) \vee W(v, \ell)$.

To check the third and fourth conditions, we require some setup. Suppose at time t_v, memory operation v happens and the *OAT* model sets $\Phi(v) = u$. Let $A = \texttt{lowest}(\texttt{readers}(t, \ell) \cap \texttt{ances}(v))$. Because the *OAT* model sets $\Phi(v) = u$, we must have $(\ell, u) \in \texttt{R}(t, A)$. Let $T = \texttt{lowest}(\texttt{xAnces}(A) \cap \texttt{writers}(t, \ell))$. By Lemma 16, we know $(\ell, u) \in \texttt{W}(t, T)$. By Theorem 17, since $(\ell, u) \in \texttt{W}(t, T)$, we know $u \in \texttt{cContent}(t, T)$. Let $X = \texttt{xLCA}(u, v)$. We must have $T \in \texttt{ances}(X)$; otherwise, we could not have $\{u, v\} \subseteq \texttt{memOps}(t, T)$.

Since $u \in \texttt{cContent}(t,T)$, we know $u \in \texttt{cContent}(t,X) \cup \texttt{oContent}(t,X)$. 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 $u \in \texttt{W}(t_v,T)$. Therefore, by Theorem 17 we know $w \in \texttt{memOps}(t_v,T)$, $w \in \texttt{aContent}(t_v,T) \cup \texttt{vContent}(t_v,T)$.

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

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

   We can show by contradiction that we must have $w \in \texttt{aContent}(t_v,Y)$. If $Y = T$, then we already know $w \in \texttt{aContent}(t_v,Y)$. Otherwise, assume $T \in \texttt{pAnces}(Y)$. If we had $w \in \texttt{cContent}(t_v,Y)$, then by Theorem 17, we must have $(\ell,y) \in \texttt{W}(t_v,Y)$. This statement contradicts the fact that *OAT* model found (ℓ,u) from transaction T, since a closer transaction Y had ℓ in its read set.

   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 Invariant 2 of Theorem 15, 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 ℓ). 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 syntax for the type system is shown below. Within the syntax, we do not differenciate classes that are Xmodule types from the classes that are not. The additional restrictions that apply to the Xmodule types are specified as one of the type rules.

For simplicity, in our type system, we make the following assumptions. First, each class has only one constructor (specified by the term *init*), and that all fields are initialized properly after the call to the constructor. Second, all field names (whether inherited or declared) are distinct. Third, the call to super is explicit. Fourth, an index is always specified when the ownership tags world and this are used. Fifth, the class names Object and Xmodule are special and assumed to be properly defined by the system. Finally, the explicit use of upcast and downcast are not allowed, as specified in the abstract syntax.

$$\begin{array}{rcl}
P & = & \mathit{defn}^*;\ e \\
\mathit{defn} & = & \textsf{class}\ \mathit{cDecl}\ \textsf{extends}\ \mathit{cDecl}\ \textsf{where}\ \mathit{constr}^*\ \{\ \mathit{field}^*;\ \mathit{init};\ \mathit{meth}^*\ \} \\
\mathit{cDecl} & = & \mathit{cn}\langle \mathit{formal}^+\rangle \mid \textsf{Object}\langle \mathit{formal}\rangle \mid \textsf{Xmodule}\langle \mathit{formal}\rangle \\
\mathit{constr} & = & \mathit{formal} < \mathit{formal} \mid \mathit{formal} = \mathit{formal} \mid \mathit{formal} \neq \mathit{formal} \\
\mathit{field} & = & t\ \mathit{fd} \\
\mathit{init} & = & \mathit{cn}\langle \mathit{formal}^+\rangle(\mathit{param}^*)\ \{\ \textsf{super}\langle \mathit{formal}^+\rangle(e^*);\ \textsf{this}.\mathit{fd} = e;^*\ \} \\
\mathit{meth} & = & t\ \mathit{mn}\langle \mathit{formal}^*\rangle(\mathit{param}^*)\ \textsf{where}\ \mathit{constr}^*\{\ e\ \} \\
\mathit{param} & = & t\ x \\
\mathit{owner} & = & \textsf{world}[i] \mid \mathit{formal} \mid \textsf{this}[i] \\
\mathit{formal} & = & f \\
t & = & \textsf{int} \mid \mathit{ct} \\
\mathit{ct} & = & \mathit{cn}\langle \mathit{owner}^+\rangle \\
e & = & \textsf{new}\ \mathit{ct}(e^*) \mid x \mid x = e \mid \textsf{let}\ (\mathit{param} = e)\ \textsf{in}\ \{\ e\ \} \mid x.\mathit{fd} \mid x.\mathit{fd} = e \mid x.\mathit{mn}\langle \mathit{owner}^+\rangle(e^*) \\
\\
\mathit{cn} & = & \textit{a class name that is not}\ \textsf{Object}\ \textit{nor}\ \textsf{Xmodule} \\
\mathit{mn} & = & \textit{a method name that is not a constructor} \\
fd & = & \textit{a field name} \\
x, y & = & \textit{a variable name} \\
f, g & = & \textit{an owner formal} \\
i, j & = & \textit{an}\ \textsf{int}\ \textit{literal}
\end{array}$$

For the constraints on owners (*constr*), the notation $<$ is used as defined in Section 3: Assuming f_1 and f_2 are instantiated with o_1 and o_2, $f_1 < f_2$ specifies that either $o_1.\mathit{name} \prec o_2.\mathit{name}$, or $o_1.\mathit{name} = o_2.\mathit{name}$ and $o_1.\mathit{index} < o_2.\mathit{index}$. Similarly, $f_1 = f_2$ specifies that $o_1.\mathit{name} = o_2.\mathit{name}$ and $o_1.\mathit{index} = o_2.\mathit{index}$. On the other hand, $f_1 \neq f_2$ specifies that either $o_1.\mathit{name} \neq o_2.\mathit{name}$, or $o_1.\mathit{name} = o_2.\mathit{name}$ and $o_1.\mathit{index} \neq o_2.\mathit{index}$.

In the type system, we use a few predicates. Before we define the predicates, we first define some notations:

Henceforth, for brevity, we use the notation $\triangleleft$ in place of the keyword `extends` (i.e. A extends B is written as $A \triangleleft B$). We also use the notation $\trianglelefteq$ between class names as the reflexive and transitive closure induced by the $\lhd$ relation. Note that the $\lhd$ is not the same as subtyping (denoted as $<:$), because $\lhd$ only considers the static relation defined by the `extends` keyword, and does not account for the ownership tags.

In addition, we define $\mathit{field} \in_d cn\langle\ldots\rangle$ to mean that class $cn\langle\ldots\rangle$ declares *field*, $\mathit{field} \in_i cn\langle\ldots\rangle$ to mean that class $cn\langle\ldots\rangle$ inherits *field*, and $\mathit{field} \in cn\langle\ldots\rangle$ to mean that either $\mathit{field} \in_d cn\langle\ldots\rangle$ or $\mathit{field} \in_i cn\langle\ldots\rangle$. We use these notations for *fd* (field name), *meth* (method), and *mn* (method name) similarly.

Now we define the predicates.

Predicate	Meaning
ClassOnce(P)	No class is declared twice in P $\forall cn, cn'$ in P, $cn \neq cn'$
FieldsOnce(P)	No class contains two fields with the same name $\forall ct\ \forall fd, fd' \in ct$ in P, $fd \neq fd'$
MethodsOnce(P)	No class declares two methods with the same name $\forall ct\ \forall mn, mn' \in_d ct$ in P, $mn \neq mn'$
WFClasses(P)	There are no cycles in the class hierarchy; i.e. the $\trianglelefteq$ relation is antisymmetric $\forall cn, cn'$ in P, $cn \trianglelefteq cn' \wedge cn' \trianglelefteq cn \implies cn = cn'$

Our typing judgment has the form: $P;\ \Gamma\ \vdash\ e : t$, where P is the program being checked to provide information about class definitions; Γ is the typing environment, providing mappings from a variable name to its static type for the free variables in e; finally, t is the static type of e.

The typing environment Γ is defined as

$$\Gamma ::= \emptyset \ |\ \Gamma,\ x : t\ \ |\ \Gamma,\ f : owner\ \ |\ \Gamma,\ constr$$

That is, the typing environtment Γ contains the types of variables, the owner parameters and the constraints among owners. When checking for well-formness of the typing environment, we assume the new entries are checking in the order listed, from left to right.

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
$P\ \vdash\ cn\langle f_{1..n}\rangle \lhd\ cn'\langle g_{1..k}\rangle$	class $cn\langle f_{1..n}\rangle$ extends class $cn'\langle g_{1..k}\rangle$
$P\ \vdash\ cn \trianglelefteq\ cn'$	cn' is an ancestor of cn in the graph defined by the extends keyword
$P\ \vdash\ \mathit{field} \in_d cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ declares *field*
$P\ \vdash\ \mathit{field} \in_i cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ inherits *field*
$P\ \vdash\ \mathit{field} \in cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ declares / inherits *field*
$P\ \vdash\ init \in cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ declares *init*
$P\ \vdash\ meth \in_d cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ declares *meth*
$P\ \vdash\ meth \in_i cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ inherits *meth*
$P\ \vdash\ meth \in cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ declares / inherits *meth*
$P;\ \Gamma\ \vdash\ \mathit{field}$	*field* is a well-formed field
$P;\ \Gamma\ \vdash\ meth$	*meth* is a well-formed method
$P;\ \Gamma\ \vdash\ wf$	typing environment Γ is well-formed
$P;\ \Gamma\ \vdash\ t$	t is a well-formed type
$P;\ \Gamma\ \vdash\ constr$	constraint *constr* is satisfied
$P;\ \Gamma\ \vdash_{owner}\ o$	o is an owner
$P;\ \Gamma\ \vdash\ e : t$	expression e has type t
$P;\ \Gamma\ \vdash\ t <:\ t'$	t is a subtype of t'

In the type rules, we also use the following auxiliary rules:

$\boxed{\textit{The Extends Relation}}$

$$\frac{P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle\ \textsf{extends}\ cn'\langle g_{1..m}\rangle\ \ldots}{P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle g_{1..m}\rangle}$$

$$\frac{}{P \vdash cn \unlhd cn} \qquad \frac{P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle g_{1..m}\rangle}{P \vdash cn \unlhd cn'} \qquad \frac{P \vdash cn \unlhd cn' \qquad P \vdash cn' \unlhd cn''}{P \vdash cn \unlhd cn''}$$

$\boxed{\textit{Type Lookup}}$

$$\begin{aligned}
&type(\,) = ()\\
&type(t\ x) = t\\
&type(t\ fd) = t\\
&type(t_1\ x_1,\ t_2,\ x_2,\ \ldots\,) = t_1,\ t_2,\ \ldots\\
&type(t\ mn\langle g_{1..k}\rangle(param^*)\{\ \ldots\ \}) = t \longrightarrow \langle g_{1..k}\rangle \longrightarrow type(param^*)
\end{aligned}$$

$\boxed{\textit{Field Lookup}}$

$$\frac{P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle\ \ldots\ \{\ \ldots\ field\ \ldots\ \}}{P \vdash field \in_d cn\langle f_{1..n}\rangle} \qquad \frac{\begin{array}{c}P \vdash field \in cn'\langle g_{1..m}\rangle\\ P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle o_{1..m}\rangle\end{array}}{P \vdash field\,[o_1/g_1]..[o_m/g_m] \in_i cn\langle f_{1..n}\rangle}$$

$$\frac{P \vdash field \in_d cn\langle f_{1..n}\rangle \vee P \vdash field \in_i cn\langle f_{1..n}\rangle}{P \vdash field \in cn\langle f_{1..n}\rangle}$$

$\boxed{\textit{Init Lookup}}$

$$\frac{P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle\ \ldots\ \{\ \ldots\ init\ \ldots\ \}}{P \vdash init \in cn\langle f_{1..n}\rangle}$$

$\boxed{\textit{Method Lookup}}$

$$\frac{P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle\ \ldots\ \{\ \ldots\ meth\ \ldots\ \}}{P \vdash meth \in_d cn\langle f_{1..n}\rangle} \qquad \frac{\begin{array}{c}P \vdash meth \in cn'\langle g_{1..m}\rangle\\ P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle o_{1..m}\rangle\end{array}}{P \vdash meth\,[o_1/g_1]..[o_m/g_m] \in_i cn\langle f_{1..n}\rangle}$$

$$\frac{P \vdash meth \in_d cn\langle f_{1..n}\rangle \vee P \vdash meth \in_i cn\langle f_{1..n}\rangle}{P \vdash meth \in cn\langle f_{1..n}\rangle}$$

$\boxed{\textit{Override Ok}}$

$$\frac{\begin{array}{c}P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle o_{1..m}\rangle\\ P \vdash meth \in_d cn\langle f_{1..n}\rangle\\ P \vdash meth \notin cn'\langle g_{1..m}\rangle\end{array}}{OverrideOk(\ cn\langle f_{1..n}\rangle,\ cn'\langle o_{1..m}\rangle,\ meth\ )} \qquad \frac{\begin{array}{c}P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle o_{1..m}\rangle\\ P \vdash meth \in_d cn\langle f_{1..n}\rangle\\ P \vdash meth' \in cn'\langle g_{1..m}\rangle\\ type(meth) = type(meth')[o_1/g_1]..[o_m/g_m]\end{array}}{OverrideOk(\ cn\langle f_{1..n}\rangle,\ cn'\langle o_{1..m}\rangle,\ meth\ )}$$

We present the type rules next.

$\boxed{\vdash P : t}$

[PROG]

$$\frac{\begin{array}{c} \mathit{WFClasses}(P) \qquad \mathit{ClassOnce}(P) \qquad \mathit{FieldsOnce}(P) \qquad \mathit{MethodsOnce}(P) \\ P = \mathit{defn}_{1..n};\ e \qquad P \vdash \mathit{defn}_i \qquad P; \emptyset \vdash e : t \end{array}}{\vdash P : t}$$

$\boxed{P \vdash \mathit{defn}}$

[CLASS]

$$\frac{\begin{array}{c} P \vdash cn \not\trianglelefteq \mathsf{Xmodule} \\ \Gamma = f_{1..n} : \mathit{owner},\ f_1 < f_i,\ \mathit{constr}^*,\ \mathsf{this} : cn\langle f_{1..n}\rangle \\ P; \Gamma \vdash \mathit{wf} \qquad P; \Gamma \vdash cn'\langle f_1,\ o^*\rangle \qquad P; \Gamma \vdash \mathit{field}_i \qquad P; \Gamma \vdash \mathit{init} \qquad P; \Gamma \vdash \mathit{meth}_i \\ \mathit{OverrideOk}(\ cn\langle f_{1..n}\rangle,\ cn'\langle f_1,\ o^*\rangle,\ \mathit{meth}_i\ ) \end{array}}{P \vdash \mathsf{class}\ \ cn\langle f_{1..n}\rangle\ \ \mathsf{extends}\ \ cn'\langle f_1,\ o^*\rangle\ \ \mathsf{where}\ \ \mathit{constr}^*\ \{\ \mathit{field}^*;\ \mathit{init};\ \mathit{meth}^*\ \}}$$

[XMODULE CLASS]

$$\frac{\begin{array}{c} P \vdash cn \trianglelefteq \mathsf{Xmodule} \\ \Gamma = f_{1..n} : \mathit{owner},\ \ f_1 < f_i,\ \ \mathit{constr}^*,\ \ \mathsf{this} : cn\langle f_{1..n}\rangle,\ \ \mathsf{this} : \mathit{owner},\ \ \mathsf{this}[i] < f_1 \\ P; \Gamma \vdash \mathit{wf} \qquad P; \Gamma \vdash cn'\langle f_1,\ o^*\rangle \qquad P; \Gamma \vdash \mathit{field}_i \qquad P; \Gamma \vdash \mathit{init} \qquad P; \Gamma \vdash \mathit{meth}_i \\ \mathit{type}(\ \mathit{field}_i\ ) \neq \mathsf{int} \qquad \mathit{OverrideOk}(\ cn\langle f_{1..n}\rangle,\ cn'\langle f_1,\ o^*\rangle,\ \mathit{meth}_i\ ) \end{array}}{P \vdash \mathsf{class}\ \ cn\langle f_{1..n}\rangle\ \ \mathsf{extends}\ \ cn'\langle f_1,\ o^*\rangle\ \ \mathsf{where}\ \ \mathit{constr}^*\ \{\ \mathit{field}^*;\ \mathit{init};\ \mathit{meth}^*\ \}}$$

$\boxed{P; \Gamma \vdash \mathit{field}}$ $\boxed{P; \Gamma \vdash \mathit{init}}$

[FIELD] [INIT]

$$\frac{P; \Gamma \vdash t}{P; \Gamma \vdash t\ \mathit{fd}} \qquad \frac{\begin{array}{c} P \vdash cn\langle f_{1..n}\rangle \triangleleft cn'\langle f_1, o_{2..m}\rangle \\ \Gamma' = \Gamma,\ \mathit{param}^* \qquad P; \Gamma' \vdash \mathit{wf} \qquad P; \Gamma' \vdash \mathsf{this}.\mathit{fd}_j = e_j \\ P \vdash \mathit{init}\langle g_{1..m}\rangle(t_i\ \ x_i\ ^{i\in 1..k}) \in cn'\langle g_{1..m}\rangle \qquad P; \Gamma' \vdash e_i : t_i\ [f_1/g_1][o_2/g_2]..[o_m/g_m] \end{array}}{P; \Gamma \vdash cn\langle f_{1..n}\rangle(\mathit{param}^*)\ \{\ \mathsf{super}\langle f_1,\ o_{2..m}\rangle(e_i\ ^{i\in 1..k});\quad \mathsf{this}.\mathit{fd} = e;^*\ \}}$$

$\boxed{P; \Gamma \vdash \mathit{meth}}$

[METHOD]

$$\frac{\begin{array}{c} \Gamma^c = f_{1..n} : \mathit{owner},\ \mathit{constr}^* \qquad P; \Gamma^c \vdash \mathit{wf} \\ \Gamma'\ =\ \Gamma,\ \Gamma^c,\ \mathit{param}^* \qquad P; \Gamma' \vdash \mathit{wf} \qquad P; \Gamma' \vdash e : t \end{array}}{P; \Gamma \vdash t\ mn\langle f_{1..n}\rangle(\mathit{param}^*)\ \mathsf{where}\ \ \mathit{constr}^*\ \{e\}}$$

$\boxed{P;\,\Gamma \vdash wf}$

[ENV $\emptyset$]

$$\frac{}{P;\,\emptyset \vdash wf}$$

[ENV X]

$$\frac{P;\,\Gamma \vdash t \qquad x \notin \mathrm{Dom}(\Gamma) \qquad P;\,\Gamma \vdash wf}{P;\,\Gamma,\, x : t \vdash wf}$$

[ENV OWNER]

$$\frac{f \notin \mathrm{Dom}(\Gamma) \qquad P;\,\Gamma \vdash wf}{P;\,\Gamma,\, f : owner \vdash wf}$$

[ENV CONSTR]

$$\frac{\begin{array}{c} constr = (o < o') \vee (o = o') \vee (o \neq o') \\ P;\,\Gamma \vdash wf \qquad P;\,\Gamma \vdash_{owner} o,\, o' \qquad \Gamma' = \Gamma,\, constr \\ \nexists_{x,y}\,(P;\,\Gamma' \vdash x < y) \wedge (P;\,\Gamma' \vdash y < x) \\ \nexists_{x,y}\,(P;\,\Gamma' \vdash x < y) \wedge (P;\,\Gamma' \vdash x = y) \qquad \nexists_{x,y}\,(P;\,\Gamma' \vdash x = y) \wedge (P;\,\Gamma' \vdash x \neq y) \end{array}}{P;\,\Gamma,\, constr \vdash wf}$$

$\boxed{P;\,\Gamma \vdash t}$

[TYPE INT]

$$\frac{}{P;\,\Gamma \vdash \mathsf{int}}$$

[TYPE OBJECT]

$$\frac{P;\,\Gamma \vdash_{owner} o}{P;\,\Gamma \vdash \mathsf{Object}\langle o \rangle}$$

[TYPE XMODULE]

$$\frac{P;\,\Gamma \vdash_{owner} o}{P;\,\Gamma \vdash \mathsf{Xmodule}\langle o \rangle}$$

[TYPE CT]

$$\frac{\begin{array}{c} P \vdash \mathsf{class}\ cn\langle f_{1..n} \rangle \ldots \mathsf{where}\ constr^{*} \ldots \\ P;\,\Gamma \vdash_{owner} o_i \qquad P;\,\Gamma \vdash o_1 < o_i \qquad P;\,\Gamma \vdash constr\,[o_1/f_1]..[o_n/f_n] \end{array}}{P;\,\Gamma \vdash cn\langle o_{1..n} \rangle}$$

$\boxed{P;\,\Gamma \vdash constr}$

[CONSTR ENV]

$$\frac{\Gamma = \Gamma',\, constr,\, \Gamma''}{P;\,\Gamma \vdash constr}$$

[< WORLD I]

$$\frac{\begin{array}{c} P;\,\Gamma \vdash_{owner} o \\ P;\,\Gamma \vdash o \neq \mathsf{world} \end{array}}{P;\,\Gamma \vdash o < \mathsf{world}[i]}$$

[< WORLD II]

$$\frac{i < j}{P;\,\Gamma \vdash \mathsf{world}[i] < \mathsf{world}[j]}$$

[< THIS]

$$\frac{\begin{array}{c} i < j \\ P;\,\Gamma \vdash_{owner} \mathsf{this} \end{array}}{P;\,\Gamma \vdash \mathsf{this}[i] < \mathsf{this}[j]}$$

[< TRANS]

$$\frac{\begin{array}{c} P;\,\Gamma \vdash o_1 < o_2 \\ P;\,\Gamma \vdash o_2 < o_3 \end{array}}{P;\,\Gamma \vdash o_1 < o_3}$$

[= WORLD]

$$\frac{i = j}{P;\,\Gamma \vdash \mathsf{world}[i] = \mathsf{world}[j]}$$

[= THIS]

$$\frac{\begin{array}{c} i = j \\ P;\,\Gamma \vdash_{owner} \mathsf{this} \end{array}}{P;\,\Gamma \vdash \mathsf{this}[i] = \mathsf{this}[j]}$$

[= TRANS]

$$\frac{\begin{array}{c} P;\,\Gamma \vdash o_1 = o_2 \\ P;\,\Gamma \vdash o_2 = o_3 \end{array}}{P;\,\Gamma \vdash o_1 = o_3}$$

[= REFL]

$$\frac{P;\,\Gamma \vdash_{owner}\ o \qquad P;\,\Gamma \vdash\ o \neq \text{world} \qquad P;\,\Gamma \vdash\ o \neq \text{this}}{P;\,\Gamma \vdash\ o = o}$$

[≠ WORLD]

$$\frac{i \neq j}{P;\,\Gamma \vdash\ \text{world}[i] \neq \text{world}[j]}$$

[≠ THIS]

$$\frac{i \neq j \qquad P;\,\Gamma \vdash_{owner}\ \text{this}}{P;\,\Gamma \vdash\ \text{this}[i] \neq \text{this}[j]}$$

[≠ WORLD]

$$\frac{P;\,\Gamma \vdash_{owner}\ \text{this}[i]}{P;\,\Gamma \vdash\ \text{this}[i] \neq \text{world}}$$

[SUBSTITUTION]

$$\frac{P;\,\Gamma \vdash\ o_1 = o_2 \qquad P;\,\Gamma \vdash\ constr}{P;\,\Gamma \vdash\ constr\ [o_1/o_2]}$$

[RELATION]

$$\frac{P;\,\Gamma \vdash\ o_1 < o_2}{P;\,\Gamma \vdash\ o_1 \neq o_2}$$

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

[OWNER WORLD]

$$\frac{}{P;\,\Gamma \vdash_{owner}\ \text{world}[i]}$$

[OWNER FORMAL]

$$\frac{\Gamma = \Gamma',\ f : owner,\ \Gamma''}{P;\,\Gamma \vdash_{owner}\ f}$$

[OWNER THIS]

$$\frac{\Gamma = \Gamma',\ \text{this} : owner,\ \Gamma''}{P;\,\Gamma \vdash_{owner}\ \text{this}[i]}$$

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

[EXP TYPE]

$$\frac{P;\,E \vdash\ t}{P;\,E \vdash\ e : t}$$

[EXP SUB]

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

[EXP NEW]

$$\frac{P;\,\Gamma \vdash\ cn\langle o_{1..n}\rangle \qquad init = cn\langle f_{1..n}\rangle(t_i\ \ x_i\ ^{i\in 1..k})\ \{\ \dots\ \} \qquad P \vdash\ init\ \in\ cn\langle f_{1..n}\rangle \qquad P;\,\Gamma \vdash\ e_i : t_i\ [o_1/f_1]..[o_n/f_n]}{P;\,\Gamma \vdash\ \text{new}\ cn\langle o_{1..n}\rangle(e_i\ ^{i\,\in\,1..k}) : cn\langle o_{1..n}\rangle}$$

[EXP VAR]

$$\frac{\Gamma\ =\ \Gamma',\ x : t,\ \Gamma''}{P;\,\Gamma \vdash\ x : t}$$

[EXP VAR ASSIGN]

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

[EXP LET]

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

[EXP REF]

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

[EXP REF ASSIGN]

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

[EXP INVOKE]

$$\frac{P \vdash\ t\ mn\langle f_{(k+1)..n}\rangle(t_i\ \ y_i\ ^{i\in 1..h})\ \ \text{where}\ \ constr^*\ \dots\ \in\ cn\langle f_{1..k}\rangle \qquad P;\,\Gamma \vdash\ x : cn\langle o_{1..k}\rangle \qquad P;\,\Gamma \vdash\ e_i : t_i\ [o_1/f_1]..[o_n/f_n] \qquad P;\,\Gamma \vdash\ constr\ [o_{k+1}/f_{k+1}]..[o_n/f_n]}{P;\,\Gamma \vdash\ x.mn\langle o_{(k+1)..n}\rangle(e_{1..h}) : t\ [o_1/f_1]..[o_n/f_n]}$$

$\boxed{P;\,\Gamma \vdash t <:\, t'}$

[SUBTYPE]

$$\frac{P;\,\Gamma \vdash cn\langle o_{1..n}\rangle \qquad P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle f^{+}\rangle}{P;\,\Gamma \vdash cn\langle o_{1..n}\rangle <:\, cn'\langle f^{+}\rangle[o_1/f_1]..[o_n/f_n]}$$

[SUBTYPE TRANS]

$$\frac{P;\,\Gamma \vdash t <:\, t' \qquad P;\,\Gamma \vdash t' <:\, t''}{P;\,\Gamma \vdash t <:\, t''}$$

[SUBTYPE REFL]

$$\frac{P;\,\Gamma \vdash t}{P;\,\Gamma \vdash t <:\, t}$$