A Programming Language Perspective on Transactional Memory Consistency

Hagit Attiya
Technion

Alexey Gotsman
IMDEA Software Institute

Sandeep Hans
Technion

Noam Rinetzky
Tel-Aviv University

ABSTRACT

Transactional memory (TM) has been hailed as a paradigm for simplifying concurrent programming. While several consistency conditions have been suggested for TM, they fall short of formalizing the intuitive semantics of atomic blocks, the interface through which a TM is used in a programming language.

To close this gap, we formalize the intuitive expectations of a programmer as *observational refinement* between TM implementations: a concrete TM observationally refines an abstract one if every user-observable behavior of a program using the former can be reproduced if the program uses the latter. This allows the programmer to reason about the behavior of a program using the intuitive semantics formalized by the abstract TM; the observational refinement relation implies that the conclusions will carry over to the case when the program uses the concrete TM. We show that, for a particular programming language and notions of observable behavior, a variant of the well-known consistency condition of opacity is sufficient for observational refinement, and its restriction to complete histories is furthermore necessary.

Our results suggest a new approach to evaluating and comparing TM consistency conditions. They can also reduce the effort of proving that a TM implements its programming language interface correctly, by only requiring its developer to show that it satisfies the corresponding consistency condition.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Software/Program Verification

Keywords

Transactional memory; atomic blocks; observational refinement

1. INTRODUCTION

Transactional memory (TM) eases the task of writing concurrent applications by letting the programmer designate certain code blocks as *atomic*. TM allows designing a program and reasoning about its correctness as if each atomic block executed as a *transaction*—in one step and without interleaving with others. As an example, Figure 1 shows how atomic blocks yield simple code for pushing an element onto a stack represented as a singly-linked list: in this case it is possible to read the top-of-the-stack pointer, point the new element to it, and change the top-of-the-stack pointer, all at once. Many TM implementations have been proposed [10], using a myriad of design approaches that, for efficiency, may execute transactions concurrently, yet aim to provide the programmer with an illusion that they are executed atomically. This illusion is not always perfect—for example, as evident from Figure 1, transactions can abort due to conflicts with concurrently running ones and need to be restarted.

```
node := new(StackNode);
node.val := val;
result := abort;
while (result == abort) do {
    result := atomic {
        node.next = Top.read();
        Top = node;
    }
}
```

Figure 1: Example of transactional memory usage

How can we be sure that a TM indeed implements atomic blocks correctly? So far, researchers have tried to achieve this through a *consistency condition* that restricts the possible TM executions. Several such conditions have been proposed, including *opacity* [8, 9], *virtual world consistency* [16], *TMS* [5, 18] and *DU-opacity* [3]. Opacity is the best-known of them; roughly speaking, it requires that for any sequence of interactions between the program and the TM, dubbed a *history*, there exist another history where:

(i) the interactions of every separate thread are the same as in the original history;

(ii) the order of non-overlapping transactions present in the original history is preserved; and

(iii) each transaction executes atomically.

Unfortunately, this definition is given from the TM's point of view, as a restriction on the set of histories it can produce, and is not connected to the semantics of a programming language. The situation for other TM consistency conditions is the same and, in fact, it is not clear which of them provide the programmer with behaviors that correspond to the intuitive notion of atomic blocks, and which of them puts the minimal restrictions on TM implementations needed to achieve this.

In this paper, we aim to bridge this gap by formalizing the intuitive expectations of a programmer as *observational refinement* [13, 14] between TM implementations. Consider two TM implementations—a *concrete* one, such as an efficient TM, and an

PODC'13, July 22–24, 2013, Montréal, Québec, Canada.

abstract one, such as a TM executing every atomic block atomically. Informally, the concrete TM observationally refines the abstract one if every behavior a user can observe of a program P linked with the concrete TM can also be observed when P is linked with the abstract TM instead. This allows the programmer to reason about the behavior of P using the intuitive semantics formalized by the abstract TM; the observational refinement relation implies that the conclusions will carry over to the case when P uses the concrete TM.

We show that one TM implementation observationally refines another if they are in an *opacity relation*. The relation requires that every history of the concrete TM have a matching history of the abstract TM satisfying the conditions (i) and (ii) above. By instantiating it with an abstract TM implementation that executes transactions atomically, we obtain the existing notion of opacity. However, our definition also allows comparing two TM implementations that execute transactions concurrently, such as a more and a less optimized one. Furthemore, we show that observational refinement between two TM implementations implies the opacity relation between thir restriction to *complete* histories, i.e., ones in which every transaction either commits or aborts.

We note that the formalization of observational refinement, and thus our results, depend on the particular choices of programming language and the notion of observations. In this first treatment of this topic, we consider a basic programming language and particular forms of observations. The features of the language are as follows:

- Threads can access shared global variables outside transactions, but not inside them. However, thread-local variables (such as node in Figure 1) can be accessed in both cases.
- An aborted transaction is not restarted automatically and modifications to thread-local variables that it may have performed are not rolled back.
- A program cannot explicitly ask the TM to abort a transaction.
- Nesting of atomic blocks is not allowed.

As observable behaviors of a program, our result allows one to take either the set of its reachable states or the set of all sequences of actions performed by its finite computations. This allows a programmer to reason about safety, but not about liveness properties.

It is likely that for other programming languages or notions of observations, other consistency criteria will be necessary or sufficient for observational refinement, resulting in different trade-offs between the efficiency of TM implementations and the flexibility of their programming interfaces (see Section 8 for discussion). We hope that the link between TM consistency conditions and programming language abstractions we establish in this paper will enable TM implementors and language designers to make informed decisions about such trade-offs. Our approach can also reduce the effort of proving that a TM implements its programming interface correctly, by only requiring its developer to show that it satisfies the corresponding consistency condition.

2. PROGRAMMING LANGUAGE

We develop our results for a simple concurrent programming language with programs consisting of a fixed, but arbitrary, number m of ***threads***, identified by $\mathsf{ThreadID} = \{1, \ldots, m\}$. Every thread $t \in \mathsf{ThreadID}$ has a private set of ***local variables*** $\mathsf{LVar}_t = \{x, y, \ldots\}$, and all threads share access to a set of ***global variables*** $\mathsf{GVar} = \{g, \ldots\}$. For simplicity, we assume that all variables are of type integer. Let $\mathsf{Var} = \mathsf{GVar} \uplus \biguplus_{t=1}^{m} \mathsf{LVar}_t$ be the set of all program variables (where $\uplus$ denotes disjoint union). In addition to variables, threads can access software or hardware transactional memory, which from now on we refer to as the ***transactional system***. The system manages a fixed collection of ***transactional objects*** $\mathsf{Obj} = \{o, \ldots\}$, each having a set of ***methods*** $\mathsf{Method} = \{f, \ldots\}$ that threads can call. For simplicity, we assume that each method takes one integer parameter and returns an integer value, and that all objects have the same set of methods.

The syntax of the language is as follows:

$$\begin{array}{lcl} C & ::= & c \mid C; C \mid \texttt{if}\,(b)\,\texttt{then}\,C\,\texttt{else}\,C \mid \\ & & \texttt{while}\,(b)\,\texttt{do}\,C \mid x := \texttt{atomic}\,\{C\} \mid x := o.f(e) \\ P & ::= & C_1 \parallel \ldots \parallel C_m \end{array}$$

where b and e denote Boolean and integer expressions over local variables, left unspecified. A program P is a parallel composition of ***sequential commands*** $C_1, \ldots, C_m$, which can include ***primitive commands*** c from a set Pcomm, sequential compositions, conditionals, loops, `atomic` blocks and object method invocations.

Primitive commands are meant to execute atomically. We do not fix their set Pcomm, but assume that it at least includes assignments to local and global variables: e.g., $g := x$. We partition the set Pcomm into $2m$ classes: $\mathsf{Pcomm} = \biguplus_{t=1}^{m}(\mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t)$. The intention is that commands from LPcomm_t can access only the local variables of thread t (LVar_t); commands from GPcomm_t can additionally access global variables ($\mathsf{LVar}_t \uplus \mathsf{GVar}$). We formalize these restriction in Section 4. We forbid a thread t from accessing local variables of other threads. Thus, the thread cannot mention such variables in the conditions of `if` and `while` commands and can only use primitive commands from $\mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t$.

An ***atomic block*** $x := \texttt{atomic}\,\{C\}$ executes the command C as a ***transaction***, which the transactional system can decide to ***commit*** or ***abort***. The system's decision is returned in the local variable x, which gets assigned distinguished values committed or aborted. We forbid nested atomic blocks and, hence, nested transactions. Inside an atomic block (and only there), the program can invoke methods on transactional objects, as in $x := o.f(e)$. Here the expression e gives the value of the method parameter, and x gets assigned the return value after the method terminates. The transactional system may decide to abort a transaction initiated by $x := \texttt{atomic}\,\{C\}$ not only upon reaching the end of the `atomic` block, but also during the execution of a method on a transactional object. Once this happens, the execution of C terminates. We do not allow programs in our language to abort a transaction explicitly. A typical pattern of using the transactional system is to execute a transaction repeatedly until it commits, as shown in Figure 1.

We forbid accessing global variables inside `atomic` blocks; thus, a thread t can use primitive commands from GPcomm_t only outside them. A transaction can use local variables of the current thread; if the transaction is aborted, these variables are not rolled back to their initial values, and the values written to them by the transaction can thus be observed by the following non-transactional code. We note that, whereas transactional objects are managed by the transactional system, global variables are not. Thus, threads can communicate via the transactional system inside `atomic` blocks, and directly via global variables outside them. We also note that the transactional system is not part of a program, but is a library used by it. Hence, the state of the transactional system is separate from the variables in Var to which the program has access.

A correct transactional system implementation has to ensure that the program behaves as though `atomic` blocks indeed execute atomically, i.e., without interleaving with actions of other threads, and that operations invoked on transactional objects in aborted transactions have no effect. This does not require the implementation to execute `atomic` blocks like this internally; it only has to

provide the illusion of their atomicity to the rest of the program. In the rest of the paper, we prove that a variant of a well-known criterion of transactional system correctness, opacity, is sufficient and necessary to validate this illusion for our programming language.

3. THE OPACITY RELATION

Histories

In this section we formalize the notion of opacity in our setting, and along the way, show how to make it more flexible. To this end, we introduce the notion of a history, which records all the interactions a program in the language of Section 2 has with the transactional system in one of its executions. A ***history***, ranged over by H and S, is a finite[1] sequence of interface actions, defined as follows.

DEFINITION 1. *An **interface action** ψ is an expression of one of the following forms:*

Request actions	*Response actions*
$(t, \mathsf{txbegin})$	(t, OK)
$(t, \mathsf{txcommit})$	$(t, \mathsf{committed}) \mid (t, \mathsf{aborted})$
$(t, \mathsf{call}\ o.f(n))$	$(t, \mathsf{ret}(n')\ o.f) \mid (t, \mathsf{aborted})$

where $t \in \mathsf{ThreadID}$, $o \in \mathsf{Obj}$, $f \in \mathsf{Method}$ and $n, n' \in \mathbb{Z}$.

Interface actions denote the control flow crossing the boundary between the program and the transactional system: request actions correspond to the control being transferred from the program to the transactional system, and response actions correspond to the control being transferred the other way around. A $(t, \mathsf{txbegin})$ action denotes a thread t requesting the transactional system to start executing a transaction; this action is generated upon entering an `atomic` block. An OK action is the only possible response by the transactional system. A txcommit action is issued when a transaction tries to commit upon exiting an `atomic` block. The transactional system responds with a committed or aborted action, depending on the result. Actions call and ret denote a call to and a return from an invocation of a method on a transactional object; they are annotated with the parameter or the return value. As we noted in Section 2, the transactional system may also decide to abort a transaction while executing a method on a transactional object. In such cases, the corresponding call action is followed by an aborted action instead of a ret one.

We use the following notation: ε is the empty history; $H(i)$ is the i-th element of a history H; $H|_t$ is the projection of H onto actions of thread t; $H|_{\neg t}$ is the projection of H onto actions of threads other than t; $H|_o$ is the projection of H onto call and ret actions on object o; $|H|$ is the length of H; $H\downarrow_i$ is the prefix of H containing i actions; $H_1 H_2$ is the concatenation of H_1 and H_2. We denote by _ an expression that is irrelevant and implicitly existentially quantified.

The interactions of programs in the language of Section 2 with the transactional system are not arbitrary; they are recorded by histories that satisfy certain well-formedness properties, summarized in the following definition.

DEFINITION 2. *A history H is **well-formed** if*

- *request and response actions are properly matched: for every thread t, $H|_t$ consists of alternating request and corresponding response actions, starting from a request action;*
- *actions denoting beginning and end of transactions are properly matched: for every thread t, in the projection of $H|_t$ to* txbegin, committed *and* aborted *actions,* txbegin *alternates with* committed *or* aborted*, starting from* txbegin*; and*
- call *and* ret *actions occur only inside transactions: for every thread t, if $H|_t = H_1\ \psi\ H_2$ for a* call *or* ret *action ψ, then $H_1 = H_1'\ \psi'\ H_1''$ for some* txbegin *action ψ', and histories H_1' and H_1'' such that H_1'' does not contain* txbegin*,* txcommit*,* committed *or* aborted *actions.*

Program executions that run utill completion are described by *complete* histories, in which every transaction either aborts or commits. This class of histories is of a particular importance to us because our necessity result holds only for this class.

DEFINITION 3. *A well-formed history H is **complete** if all transactions in it have completed: if $H = H_1\ (t, \mathsf{txbegin})\ H_2$, then H_2 contains a $(t, \mathsf{committed})$ or $(t, \mathsf{aborted})$ action.*

We specify the behavior of a transactional system implementation by the set of possible interactions it can have with its clients—its ***history set*** $\mathcal{H}$, which is a prefix-closed set of well-formed histories. For our purposes, this specifies the behaviour of a transactional system completely; thus, in the following, we often conflate the notion of a transactional system and its history set.

We denote the ***complete subset*** of a history set $\mathcal{H}$ by

$$\mathcal{H}|_{\mathsf{complete}} = \{H \mid H \in \mathcal{H} \wedge (H \text{ is complete})\}.$$

The definition of the opacity relation

We define the notion of opacity in a slightly more flexible way than the original one [8, 9], inspired by the approach taken when defining the correctness of concurrent libraries via linearizability [15]. Namely, we define the correctness of a transactional system implementation by relating its history set to that of an ***abstract*** implementation, whose behavior it has to simulate; in this context, we call the original implementation ***concrete***. The abstract implementation is typically one in which `atomic` blocks actually execute atomically and methods called by aborted transactions have no effect. As we show below, we can obtain the original definition of opacity by instantiating ours with such an abstract implementation; however, our definition can also be used to compare two arbitrary implementations. To disambiguate, in the following we refer to our notion as the ***opacity relation***, instead of just opacity.

According to the following definition, a concrete transactional system $\mathcal{H}_C$ is in the opacity relation with an abstract transactional system $\mathcal{H}_A$, if every history H from $\mathcal{H}_C$ can be matched by a history S from $\mathcal{H}_A$ that "looks similar" to H from the perspective of the program. The similarity is formalized by a relation $H \sqsubseteq S$, which requires S to be a permutation of H preserving the order of actions within a thread and that of non-overlapping transactions (whether committed or aborted). Here the duration of a transaction is defined by the interval from its txbegin action to the corresponding committed or aborted action (or to the end of the history if there is none).

DEFINITION 4. *A well-formed history H is in the **opacity relation** with a well-formed history S, denoted $H \sqsubseteq S$, if there is a bijection $\theta : \{1, \ldots, |H|\} \to \{1, \ldots, |S|\}$ such that $\forall i.\, H(i) = S(\theta(i))$ and*

$$\begin{aligned}\forall i, j.\, i < j \wedge ((\exists t.\, H(i) = (t, \_) \wedge H(j) = (t, \_)) \vee \\ (H(i) \in \{(\_, \mathsf{committed}), (\_, \mathsf{aborted})\} \wedge \\ H(j) = (\_, \mathsf{txbegin}))) \\ \implies \theta(i) < \theta(j).\end{aligned}$$

[1] We do not consider infinite computations in this paper; see Section 8 for a discussion.

*A transactional system $\mathcal{H}_C$ is in the **opacity relation** with a transactional system $\mathcal{H}_A$, denoted $\mathcal{H}_C \sqsubseteq \mathcal{H}_A$, if*

$$\forall H \in \mathcal{H}_C.\, \exists S \in \mathcal{H}_A.\, H \sqsubseteq S.$$

In the following, for $i < j$ we say that the actions $H(i)$ and $H(j)$ are in the ***per-thread order*** in H when they are by the same thread and in the ***real-time order*** when $H(i) = (\_, \mathsf{committed})$ or $H(i) = (\_, \mathsf{aborted})$ and $H(j) = (\_, \mathsf{txbegin})$. Thus, $H \sqsubseteq S$ requires that the per-thread and real-time orders between actions in H be preserved in S. We now make some comments concerning the choices taken in this definition.

- As we show in Section 7, preserving the real-time order in Definition 4 is necessary to validate observational refinement due to the fact that, in our programming language, threads can access global variables outside transactions and, hence, can notice the order of non-overlapping transactions.
- Definition 4 treats committed and aborted transactions uniformly. This is a characteristic feature of opacity: it ensures that the results returned by methods of transactional objects inside aborted transactions are as consistent as those returned inside committed transactions (we return to this point in Sections 7 and 8).
- The abstract history S in Definition 4 is not required to be sequential, i.e., it may have overlapping executions of transactions. This allows the definition to compare behaviors of two realistic transactional system implementations that actually execute transactions concurrently. We also allow H to contain uncompleted transactions (without a final committed or aborted action) arising, e.g., because the corresponding thread has been preempted. In this case, we require the same behavior to be reproduced in the matching history S, which is possible because the latter does not have to be sequential [7].

Comparison with the original opacity definition

We now show how the original notion of opacity [8, 9] can be obtained from ours by instantiating Definition 4 with an abstract transactional system $\mathcal{H}_{\mathsf{atomic}}$ in which `atomic` blocks execute atomically and methods called by aborted transactions have no effect. We first introduce the ingredients needed to define this system. We start by defining a special class of *non-interleaved* histories.

DEFINITION 5. *A well-formed history H is **non-interleaved** if actions by any two transactions do not overlap: if $H = H_1\,(t, \mathsf{txbegin})\,H_2\,(t', \mathsf{txbegin})\,H_3$, where H_2 does not contain* txbegin *actions, then either H_2 contains a $(t, \mathsf{committed})$ or a $(t, \mathsf{aborted})$ action, or there are no actions by thread t in H_3.*

Note that a non-interleaved history does not have to be complete. In fact, the history set $\mathcal{H}_{\mathsf{atomic}}$ we are about to define contains only non-interleaved histories, but some of them are incomplete. This is because a concrete transactional system may produce histories with incomplete transactions, and our opacity relation requires these transactions to stay incomplete in the matching history of the abstract system. To check whether an incomplete history should be included into $\mathcal{H}_{\mathsf{atomic}}$, we first complete it with the aid of the operation defined below, which aborts every transaction that has not tried to commit yet and commits or aborts every transaction that has tried to commit (as witnessed by a txcommit action), but has not yet got a response.

DEFINITION 6. *A history H is a **completing history** for an interface action ψ, if the following holds:*

- *if $\psi = (t, \mathsf{call}\ o.f(n))$, then $H = (t, \mathsf{aborted})$;*
- *if $\psi = (t, \mathsf{txbegin})$, then $H = (t, \mathsf{OK})\,(t, \mathsf{txcommit})\,(t, \mathsf{aborted})$;*
- *if $\psi \in \{(t, \mathsf{ret}(n)\ o.f), (t, \mathsf{OK})\}$, then $H = (t, \mathsf{txcommit})\,(t, \mathsf{aborted})$;*
- *if $\psi = (t, \mathsf{txcommit})$, then $H = (t, \mathsf{committed})$ or $H = (t, \mathsf{aborted})$;*
- *otherwise, $H = \varepsilon$.*

DEFINITION 7. *A history H_c is a **non-interleaved completion** of a non-interleaved history H, if H_c is a non-interleaved complete history that can be constructed from H by adding a completing history for the last action of every thread right after this action. We denote the set of non-interleaved completions of H by $\mathsf{nicomp}(H)$.*

To define $\mathcal{H}_{\mathsf{atomic}}$, we also need to know the intended semantics of operations on transactional objects. We describe the semantics for an object $o \in \mathsf{Obj}$ by fixing all sequences of actions on o that are considered correct when executed by a sequential program. More precisely, a ***sequential specification*** of an object o is a set of histories $[\![o]\!]$ such that:

- $[\![o]\!]$ is prefix-closed;
- each $H \in [\![o]\!]$ consists of alternating call and ret actions on o, starting from a call action, where every ret is by the same thread as the preceding call; and
- $[\![o]\!]$ is insensitive to thread identifiers: for any $H \in [\![o]\!]$, changing the thread identifier in call-ret pair of adjacent actions in H yields a history in $[\![o]\!]$.

For example, $[\![o]\!]$ for a register object o would consist of histories where each read method invocation returns the value written by the latest preceding write method invocation (or the default value if there is none).

Using sequential specifications for all objects, we now define when a complete and non-interleaved history H respects the object semantics. Let $H(i)$ be a call or ret action on an object o. We say that $H(i)$ is ***legal*** in H if $H'|_o \in [\![o]\!]$, where H' is the history obtained from H by projecting $H\downarrow_i$ on all actions by committed transactions and the transaction containing $H(i)$. A complete and non-interleaved history H is ***legal*** if all call and ret actions in H are legal. We now let $\mathcal{H}_{\mathsf{atomic}}$ to be the set of all non-interleaved histories that can be completed to a legal history:

$$\mathcal{H}_{\mathsf{atomic}} = \{H \mid H \text{ is non-interleaved} \wedge \exists \text{ legal } H_c \in \mathsf{nicomp}(H)\}.$$

Thus, we can say that a transactional system $\mathcal{H}_C$ establishes the illusion of atomicity for transactions if $\mathcal{H}_C \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$. Note that, since we require the history set of a transactional system to be prefix-closed, this criterion checks every prefix of any history produced by $\mathcal{H}_C$, just like opacity as formulated in [9]. However, in other aspects this definition is of a different form than opacity, which we can formulate in our setting as follows.

DEFINITION 8. *A history H_c is a **suffix completion** of a history H, if it is a complete history, H is a prefix of H_c, and H_c can be constructed from H by appending to it a completing history for the last action of every thread. We denote the set of suffix-completions of H by $\mathsf{scomp}(H)$.*

DEFINITION 9. *A transactional system $\mathcal{H}_C$ is **opaque** if for every history $H \in \mathcal{H}_C$, there exists a history $H_c \in \mathsf{scomp}(H)$ and a complete, non-interleaved and legal history S_c such that $H_c \sqsubseteq S_c$.*

The main difference is that Definition 9 first completes a history from $\mathcal{H}_C$ and then finds its match according to the opacity relation;

our criterion $\mathcal{H}_C \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$ first finds the match and then completes the matching history. For technical reasons, Definition 9 also uses a slightly different completion, putting completing histories at the end to avoid creating new real-time orderings.

Fortunately, completion and matching commute, and thus the two formulations of opacity are equivalent.

PROPOSITION 10. *A transactional system $\mathcal{H}_C$ is opaque if and only if $\mathcal{H}_C \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$.*

We prove the proposition in [2, Appendix A]. The formulation $\mathcal{H}_C \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$ is more convenient for us, since the statement of observational refinement we give in Section 5 below requires us to leave the histories of the concrete transactional system intact. Stating transactional system consistency in this way also avoids the need to bake in completions (Definition 7) into the definition of the opacity relation (Definition 4): the treatment of incomplete transactions can be deferred to the choice of the abstract transactional system.

4. PROGRAMMING LANGUAGE SEMANTICS

Our goal is to establish a connection between conditions on transactional system implementations, such as the opacity relation from Section 3, and the behavior of programs that use these systems, such as those in the language of Section 2. To this end, in this section we define the *semantics* of our programming language, i.e., what kinds of computations can result when a program executes with a particular transactional system.

A program computation is captured by a ***trace*** τ, which is a finite sequence of *actions*, each describing a single computation step.

DEFINITION 11. *An **action** φ is an expression of one of the following forms: $\varphi ::= \psi \mid (t, c)$, where $t \in \mathsf{ThreadID}$ and $c \in \mathsf{Pcomm}$. We denote that set of all actions by* Action.

In addition to interface actions, we have actions of the form (t, c), which denote the execution of a primitive command by thread t. To denote the evaluation of conditions in `if` and `while` statements, we assume that the sets LPcomm_t contain special primitive commands $\mathsf{assume}(b)$, where b is a Boolean expression over local variables of thread t, defining the condition. We state their semantics formally below; informally, $\mathsf{assume}(b)$ does nothing if b holds in the current program state, and stops the computation otherwise. Thus, it allows the computation to proceed only if b holds. The assume commands are only used in defining the semantics of the programming language; hence, we forbid threads from using them directly.

We denote by $\mathsf{history}(\tau)$ the history obtained by projecting a trace τ to interface actions. We use various operations on histories defined in Section 3 for traces as well. As is the case for histories, programs in the language of Section 2 do not generate arbitrary traces, but only those satisfying certain conditions summarized in the following definition.

DEFINITION 12. *A trace τ is **well-formed** if*

- *the history* $\mathsf{history}(\tau)$ *is well-formed;*
- *thread t does not access local variables of other threads: if $\tau = \tau_1\,(t, c)\,\tau_2$, then $c \in \mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t$; and*
- *commands in τ do not access global variables inside a transaction: if $\tau = \tau_1\,(t, c)\,\tau_2$ for $c \in \mathsf{GPcomm}_t$, then it is not the case that $\tau_1 = \tau_1'\,(t, \mathsf{txbegin})\,\tau_1''$, where τ_1'' does not contain* committed *or* aborted *actions.*

We denote the set of well-formed traces by WTrace.

We use two additional operations on traces. For a trace τ, we define $\mathsf{trans}(\tau)$ and $\mathsf{nontrans}(\tau)$ as the subsequence of actions in τ executed inside transactions (including txbegin, committed and aborted actions), respectively, outside them (excluding txbegin, committed and aborted actions). Formally, we include an action $\varphi = (t, \_)$ such that $\tau|_t = \tau_1\,\varphi\,\tau_2$ into $\mathsf{trans}(\tau)$ if:

- φ is a txbegin, committed or aborted action; or
- $\tau_1 = \tau_1'\,(t, \mathsf{txbegin})\,\tau_1''$, where τ_1'' does not contain committed or aborted actions.

All other actions form $\mathsf{nontrans}(\tau)$. Actions in τ that are in $\mathsf{trans}(\tau)$ are ***transactional*** and all others are ***non-transactional***.

A ***state*** of a program records the values of all its variables: $s \in \mathsf{State} = \mathsf{Var} \to \mathbb{Z}$. The semantics of a program $P = C_1 \parallel \cdots \parallel C_m$ is given by the set of traces $[\![P]\!](s, \mathcal{H}) \in \mathcal{P}(\mathsf{WTrace})$ it produces when run with a transactional system $\mathcal{H}$ from an initial state s. We define this set in two stages. First, we define the set $[\![P]\!](s) \in \mathcal{P}(\mathsf{WTrace})$ that a program produces when run from s with the behaviors of the transactional system unrestricted. We then compute the set of traces produced by P when run with a given transactional system $\mathcal{H}$ by selecting those traces that interact with the transactional system in a way consistent with $\mathcal{H}$:

$$[\![P]\!](s, \mathcal{H}) = \{\tau \mid \tau \in [\![P]\!](s) \wedge \mathsf{history}(\tau) \in \mathcal{H}\}. \qquad (1)$$

The set $[\![P]\!](s)$ is itself computed in two stages[2]. First, we compute a trace set $A(P) \in \mathcal{P}(\mathsf{WTrace})$ that resolves all issues regarding sequential control flow and interleaving. Intuitively, if one thinks of each sequential command C_t in P as a control-flow graph, then $A(P)$ contains all possible interleavings of paths in the control-flow graph of all the commands C_t starting from their source nodes. The set $A(P)$ is a superset of all the traces that can actually be executed: e.g., if a thread executes the command

$$x := 1;\ \texttt{if}\ (x = 1)\ y := 1\ \texttt{else}\ y := 2 \qquad (2)$$

where x is a local variable, then $A(P)$ will contain a trace where $y := 2$ is executed instead of $y := 1$. To filter out such nonsensical traces, we *evaluate* every trace to determine whether its control flow is consistent with the expected behavior of its actions. This is formalized by a function $\mathsf{eval} : \mathsf{State} \times \mathsf{WTrace} \to \mathcal{P}(\mathsf{State})$ that, given an initial state and a trace, produces the set of states resulting from executing the actions in the trace, or an empty set if the trace is infeasible. Then we let

$$[\![P]\!](s) = \{\tau \mid \tau \in A(P) \wedge \mathsf{eval}(s, \tau) \neq \emptyset\}. \qquad (3)$$

The rest of this section defines the trace set $A(P)$ and the evaluation function eval formally. The definitions follow the intuitive semantics of our programming language and can be skipped on first reading (they are only used in the proofs of Lemmas 18 and 20 in Section 6 and in the detailed proof of Theorem 24 in [2, Appendix B]).

Trace set $A(P)$

The function $A(\cdot)$ in Figure 2 maps commands and programs to the set of their possible traces. $A(C)t$ gives the set of traces produced by a command C when it is executed by thread t. To define $A(P)$, we first compute the set of all the interleavings of traces produced by the threads constituting P. Formally, $\tau \in \mathsf{interleave}(\tau_1, \ldots, \tau_m)$ if and only if every action in τ is performed by some thread $t \in \{1, \ldots, m\}$, and $\tau|_t = \tau_t$ for every thread $t \in \{1, \ldots, m\}$. We then let $A(P)$ be the set of all prefixes of the resulting traces, as denoted by the prefix operator. We take prefix

[2]Here we define the set $[\![P]\!](s)$ in a denotational style; a definition using structural operational semantics would also be appropriate.

$$
\begin{array}{rcl}
A(c)t & = & \{(t, c)\} \\
A(C_1; C_2)t & = & \{\tau_1\, \tau_2 \mid \tau_1 \in A(C_1)t \wedge \tau_2 \in A(C_2)t\} \\
A(\texttt{if}\ (b)\ \texttt{then}\ C_1\ \texttt{else}\ C_2)t & = & \{(t, \mathsf{assume}(b))\, \tau_1 \mid \tau_1 \in A(C_1)t\} \cup \{(t, \mathsf{assume}(\neg b))\, \tau_2 \mid \tau_2 \in A(C_2)t\} \\
A(\texttt{while}\ (b)\ \texttt{do}\ C)t & = & \{((t, \mathsf{assume}(b))\, (A(C)t))^*\, (t, \mathsf{assume}(\neg b))\} \\
A(x := o.f(e))t & = & \{(t, \mathsf{assume}(e = n))\, (t, \mathsf{call}\ o.f(n))\, (t, \mathsf{ret}(n')\ o.f)\, (t, x := n') \mid n, n' \in \mathbb{Z}\} \cup \\
 & & \{(t, \mathsf{assume}(e = n))\, (t, \mathsf{call}\ o.f(n))\, (t, \mathsf{aborted}) \mid n \in \mathbb{Z}\} \\
A(x := \texttt{atomic}\ \{C\})t & = & \{(t, \mathsf{txbegin})\, (t, \mathsf{OK})\, \tau\, (t, \mathsf{aborted})\, (t, x := \mathsf{aborted}) \mid \tau\, (t, \mathsf{aborted}) \in A(C)t\} \cup \\
 & & \{(t, \mathsf{txbegin})\, (t, \mathsf{OK})\, \tau\, (t, \mathsf{txcommit})\, (t, \mathsf{committed})\, (t, x := \mathsf{committed}) \mid \tau \in A(C)t \wedge \tau \neq \_\, (t, \mathsf{aborted})\} \cup \\
 & & \{(t, \mathsf{txbegin})\, (t, \mathsf{OK})\, \tau\, (t, \mathsf{txcommit})\, (t, \mathsf{aborted})\, (t, x := \mathsf{aborted}) \mid \tau \in A(C)t \wedge \tau \neq \_\, (t, \mathsf{aborted})\} \\
A(C_1 \parallel \ldots \parallel C_m) & = & \mathsf{prefix}(\bigcup\{\mathsf{interleave}(\tau_1, \ldots, \tau_m) \mid \forall t.\, 1 \leq t \leq m \implies \tau_t \in A(C_t)t\})
\end{array}
$$

Figure 2: The function $A(\cdot)$ mapping commands and programs to the set of all their possible traces

closure here to account for incomplete program computations as well as those in which the scheduler preempts a thread forever.

$A(c)t$ returns a singleton set with the action corresponding to the primitive command c (recall that primitive commands execute atomically). $A(C_1; C_2)t$ concatenates all possible traces corresponding to C_1 with those corresponding to C_2. The set of traces for a conditional considers cases where either branch is taken. We record the decision using an assume action; at the evaluation stage, this allows us to ensure that this decision is consistent with the program state. The trace set for a loop is defined using the Kleene closure operator * to produce all possible unfoldings of the loop body. Again, we record branching decisions using assume actions.

The trace set of a method invocation $x := f(e)$ includes both traces where the method executes successfully and where the current transaction is aborted. The former set is constructed by non-deterministically choosing two integers n and n' to describe the parameter n and the return value n' for the method call. To ensure that e indeed evaluates to n, we insert $\mathsf{assume}(e = n)$ before the call action, and to ensure that x gets the return value n', we add the assignment $x := n'$ after the ret action. Note that some of the choices here might not be feasible: the chosen n might not be the value of the parameter expression e when the method is invoked, or the method might never return n' when called with n. Such infeasible choices are filtered out at the following stages of the semantics definition: the former at the evaluation stage (3) by the semantics of assume, and the latter in (1) by selecting the traces from $[\![P]\!](s)$ that interact with the transactional system correctly.

The trace set of $x := \texttt{atomic}\ \{C\}$ contains traces in which C is aborted in the middle of its execution (at an object operation) and those in which C executes until completion and then the transaction commits or aborts. From the restrictions on programs introduced in Section 2, we immediately get:

PROPOSITION 13. *For any program P, the set $A(P)$ contains only well-formed traces.*

Semantics of primitive commands

To define evaluation, we assume a semantics of every command $c \in \mathsf{Pcomm}$, given by a function $[\![c]\!]$ that defines how the program state is transformed by executing c. As we noted in Section 2, different classes of primitive commands are supposed to access only certain subsets of variables. To ensure that this is indeed the case, we define $[\![c]\!]$ as a function of only those variables that c is allowed to access. Namely, the semantics of $c \in \mathsf{LPcomm}_t$ is given by

$$[\![c]\!] : (\mathsf{LVar}_t \to \mathbb{Z}) \to \mathcal{P}(\mathsf{LVar}_t \to \mathbb{Z}),$$

and the semantics of $c \in \mathsf{GPcomm}_t$, by

$$[\![c]\!] : ((\mathsf{LVar}_t \uplus \mathsf{GVar}) \to \mathbb{Z}) \to \mathcal{P}((\mathsf{LVar}_t \uplus \mathsf{GVar}) \to \mathbb{Z}).$$

For a valuation q of variables that c is allowed to access, $[\![c]\!](q)$ yields the set of their valuations that can be obtained by executing c from a state with variable values q. Note that this allows c to be non-deterministic. For example, an assignment command $x := g$ has the following semantics:

$$[\![x := g]\!](q) = q[x \mapsto q(g)],$$

where $q[x \mapsto q(g)]$ is the function that has the same value as q everywhere, except for x, where it has the value $q(g)$. We define the semantics of assume commands following the informal explanation given at the beginning of this section: for example,

$$[\![\mathsf{assume}(x = n)]\!](q) = \begin{cases} \{q\}, & \text{if } q(x) = n; \\ \emptyset, & \text{otherwise.} \end{cases} \tag{4}$$

Thus, when the condition in assume does not hold of q, the command stops the computation by not producing any output states.

We lift functions $[\![c]\!]$ to full states by keeping the variables that c is not allowed to access unmodified. For example, if $c \in \mathsf{GPcomm}_t$, then for all $u \in \mathsf{Var}$ we let

$$[\![c]\!](s)(u) = \begin{cases} [\![c]\!](s|_{\mathsf{LVar}_t \uplus \mathsf{GVar}})(u), & \text{if } u \in \mathsf{LVar}_t \uplus \mathsf{GVar}; \\ s(u), & \text{otherwise.} \end{cases}$$

where $s|_{\mathsf{LVar}_t \uplus \mathsf{GVar}}$ is the restriction of s to variables in $\mathsf{LVar}_t \uplus \mathsf{GVar}$.

Trace evaluation

Using the semantics of primitive commands, we first define the evaluation of a single action on a given state:

$$\begin{array}{rcl} \mathsf{eval} : \mathsf{State} \times \mathsf{Action} \to \mathcal{P}(\mathsf{State}) & & \\ \mathsf{eval}(s, (t, c)) & = & [\![c]\!](s); \\ \mathsf{eval}(s, \varphi) & = & \{s\} \text{ for all other actions } \varphi. \end{array}$$

Note that this does not change the state s as a result of interface actions, since their return values are assigned to local variables by separate actions introduced when generating $A(P)$. We then lift eval to traces as follows:

$$\begin{array}{rcl} \mathsf{eval} : \mathsf{State} \times \mathsf{WTrace} \to \mathcal{P}(\mathsf{State}) & & \\ \mathsf{eval}(s, \varepsilon) & = & \{s\}; \\ \mathsf{eval}(s, \tau\varphi) & = & \{s'' \mid \exists s'.\, s' \in \mathsf{eval}(s, \tau) \wedge s'' \in \mathsf{eval}(s', \varphi)\}. \end{array}$$

This allows us to define $[\![P]\!](s)$ as the set of those traces from $A(P)$ that can be evaluated from s without getting stuck, as formalized

by (3). Note that this definition enables the semantics of assume defined by (4) to filter out traces that make branching decisions inconsistent with the program state. For example, consider the program (2). The set $A(P)$ includes traces where both branches are explored. However, due to the semantics of the assume actions added to the traces according to Figure 2, only the trace executing $y := 1$ will result in a non-empty set of final states after the evaluation and, therefore, only this trace will be included into $[\![P]\!](s)$.

5. OBSERVATIONAL REFINEMENT

Informally, a concrete transactional system $\mathcal{H}_C$ *observationally refines* an abstract transactional system $\mathcal{H}_A$, if replacing $\mathcal{H}_C$ by $\mathcal{H}_A$ in a program leaves all its original user-observable behaviors reproducible. The formal definition depends on which aspects of program behavior we consider observable. One possibility is to allow the user to observe the values of all variables during the program execution. The corresponding notion of observational refinement is stated as follows.

DEFINITION 14. *A transactional system $\mathcal{H}_C$ **observationally refines** a transactional system $\mathcal{H}_A$ **with respect to states**, denoted by $\mathcal{H}_C \preceq_{\mathsf{state}} \mathcal{H}_A$, if*

$$\forall P.\, \forall s.\, \forall \tau \in [\![P]\!](s, \mathcal{H}_C).\, \exists \tau' \in [\![P]\!](s, \mathcal{H}_A).\ \mathsf{eval}(s, \tau) = \mathsf{eval}(s, \tau').$$

Note that, since the semantics of a program P includes traces corresponding to its incomplete computations (see the use of prefix-closure in Figure 2), we allow observing intermediate program states as well as final ones. Definition 14 implies that, if a program using the abstract transactional system $\mathcal{H}_A$ satisfies a correctness property stated in terms of such states, then it will still satisfy the property if it uses the concrete system $\mathcal{H}_C$ instead.

In some situations, we may also be interested in observing separate program actions and the order between them, rather than the set of all reachable program states. In particular, this is desirable for checking the validity of linear-time temporal properties over program traces. We formulate the corresponding notion of observational refinement as follows.

DEFINITION 15. *Well-formed traces τ and τ' are **observationally equivalent**, written $\tau \sim \tau'$, if*

$$(\forall t \in \mathsf{ThreadID}.\, \tau|_t = \tau'|_t) \wedge (\mathsf{nontrans}(\tau) = \mathsf{nontrans}(\tau')).$$

*A transactional system $\mathcal{H}_C$ **observationally refines** a transactional system $\mathcal{H}_A$ **with respect to traces**, denoted by $\mathcal{H}_C \preceq_{\mathsf{trace}} \mathcal{H}_A$, if*

$$\forall P.\, \forall s.\, \forall \tau \in [\![P]\!](s, \mathcal{H}_C).\, \exists \tau' \in [\![P]\!](s, \mathcal{H}_A).\, \tau \sim \tau'.$$

Traces related by $\sim$ are thus considered indistinguishable to the user, and, hence, the user can observe which equivalence class over $\sim$ the trace executed by the program belongs to.

We are now in a position to state our main result.

THEOREM 16.

$$\mathcal{H}_C \sqsubseteq \mathcal{H}_A \implies \mathcal{H}_C \preceq_{\mathsf{trace}} \mathcal{H}_A \implies \mathcal{H}_C \preceq_{\mathsf{state}} \mathcal{H}_A \implies \mathcal{H}_C|_{\mathsf{complete}} \sqsubseteq \mathcal{H}_A|_{\mathsf{complete}}.$$

Thus, for our programming language, the opacity relation implies observational refinement with respect to traces or states, and either of these implies that the complete subsets of the transactional systems are in the opacity relation.

The first and the second implications from Theorem 16 are proved in Section 6, and the third one, in Section 7.

6. SUFFICIENCY OF THE OPACITY RELATION

Our goal in this section is to show that the opacity relation implies observational refinement with respect to traces. The next lemma (proved below) is key in establishing this: it shows that a trace τ_H with a history H can be transformed into an equivalent trace τ_S with a history S that is in the opacity relation with H.

LEMMA 17 (REARRANGEMENT). *For any well-formed histories H and S:*

$$H \sqsubseteq S \implies (\forall \textit{well-formed } \tau_H.\, \mathsf{history}(\tau_H) = H \implies \exists \textit{well-formed } \tau_S.\, \mathsf{history}(\tau_S) = S \wedge \tau_H \sim \tau_S).$$

We also rely on the following lemma, which straightforwardly implies that observational refinement with respect to traces implies that with respect to states. The proof of the lemma (given below) relies on the restrictions on accesses to variables in Definition 12.

LEMMA 18. *For all well-formed traces τ_H and τ_S,*

$$\tau_H \sim \tau_S \wedge \mathsf{eval}(s, \tau_H) \neq \emptyset \implies \forall s.\, \mathsf{eval}(s, \tau_H) = \mathsf{eval}(s, \tau_S).$$

COROLLARY 19. $\mathcal{H}_C \preceq_{\mathsf{trace}} \mathcal{H}_A \implies \mathcal{H}_C \preceq_{\mathsf{state}} \mathcal{H}_A$.

Lemma 18 also allows us to conclude that the trace τ_S resulting from the transformation in Lemma 17 can be produced by a program P if so can the original trace τ_H.

LEMMA 20. *If $\tau_H \in [\![P]\!](s)$ and $\tau_H \sim \tau_S$, then $\tau_S \in [\![P]\!](s)$.*

PROOF. Let $P = C_1 \parallel \ldots \parallel C_m$. Consider s, τ_H and τ_S such that $\tau_H \in [\![P]\!](s)$ and $\tau_H \sim \tau_S$. Then for some τ' we have $\tau_H\, \tau' \in A(P)$ and $\mathsf{eval}(s, \tau_H) \neq \emptyset$. This implies $(\tau_H\, \tau')|_t \in A(C_t)t$ for any thread t. Since $\tau_H \sim \tau_S$, we have that, $\tau_S|_t = \tau_H|_t$. Then $(\tau_S\, \tau')|_t \in A(C_t)t$ and by the definition of $A(P)$ in Figure 2 we get $\tau_S \in A(P)$. Furthermore, by Lemma 18, $\mathsf{eval}(s, \tau_S) \neq \emptyset$, so that $\tau_S \in [\![P]\!](s)$. □

THEOREM 21. $\mathcal{H}_C \sqsubseteq \mathcal{H}_A \implies \mathcal{H}_C \preceq_{\mathsf{trace}} \mathcal{H}_A$.

PROOF. Assume $\mathcal{H}_C \sqsubseteq \mathcal{H}_A$ and consider a program P, a state s and a trace $\tau_H \in [\![P]\!](s, \mathcal{H}_C)$. Let $\mathsf{history}(\tau_H) = H$; then $\tau_H \in [\![P]\!](s)$ and $H \in \mathcal{H}_C$. Since $\mathcal{H}_C \sqsubseteq \mathcal{H}_A$, there exists a history $S \in \mathcal{H}_A$ such that $H \sqsubseteq S$. Since H and S are well-formed, Lemma 17 implies that there is a well-formed trace τ_S such that $\tau_H \sim \tau_S$ and $\mathsf{history}(\tau_S) = S$. Then by Lemma 20 we have $\tau_S \in [\![P]\!](s)$. Since $S \in \mathcal{H}_A$, this implies $\tau_S \in [\![P]\!](s, \mathcal{H}_A)$. □

Proof of Lemma 17 (Rearrangement)

Consider H, S and τ_H such that $H \sqsubseteq S$ and $\mathsf{history}(\tau_H) = H$. Note that $|H| = |S|$. To obtain the desired trace τ_S, we inductively construct a sequence of well-formed traces τ^i, $i = 0..|S|$ with well-formed histories $H^i = \mathsf{history}(\tau^i)$ such that

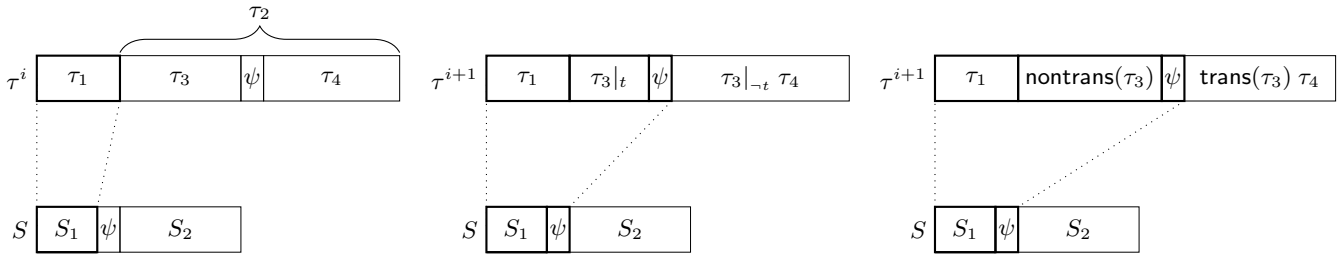
$$H^i\downarrow_i = S\downarrow_i;\quad H^i \sqsubseteq S;\quad \tau_H \sim \tau^i. \tag{5}$$

We then let $\tau_S = \tau^{|S|}$, so that $\tau_H \sim \tau^{|S|}$ and

$$\mathsf{history}(\tau^{|S|}) = H^{|S|} = H^{|S|}\downarrow_{|S|} = S\downarrow_{|S|} = S,$$

as required. Note that the condition $H^i \sqsubseteq S$ in (5) is not used to establish the required properties of τ_S; we add it so that the induction goes through.

We start the construction of the sequence of traces τ^i with $\tau^0 = \tau_H$, so that $H^0 = H$ and all the requirements in (5) hold trivially. Assume a trace τ^i satisfying (5) was constructed; we get τ^{i+1} from τ^i by applying the following lemma. Since $\sim$ is transitive, the conclusion of the lemma implies the desired properties of τ^{i+1}.



(a) Trace τ^i and history S (b) Case of $\psi \neq (t, \mathsf{txbegin})$ (c) Case of $\psi = (t, \mathsf{txbegin})$

Figure 3: An illustration of the transformations performed in the proof of Lemma 22

LEMMA 22. *Assume well-formed histories H^i and S and a well-formed trace τ^i such that*

$$\mathsf{history}(\tau^i) = H^i; \quad H^i\downarrow_i = S\downarrow_i; \quad H^i \sqsubseteq S.$$

Then there exist some well-formed history H^{i+1} and a well-formed trace τ^{i+1} such that

$$\mathsf{history}(\tau^{i+1}) = H^{i+1}; \quad H^{i+1}\downarrow_{i+1} = S\downarrow_{i+1}; \quad H^{i+1} \sqsubseteq S; \quad \tau^i \sim \tau^{i+1}.$$

PROOF. Let $S = S_1\,\psi\,S_2$, where $|S_1| = i$. By assumption, $\mathsf{history}(\tau^i)\downarrow_i = H^i\downarrow_i = S\downarrow_i = S_1$. Thus, for some traces τ_1 and τ_2, we have $\tau^i = \tau_1\,\tau_2$, where τ_1 is the minimal prefix of τ^i such that $\mathsf{history}(\tau_1) = S_1$. We also have $H^i \sqsubseteq S$, and, hence, there exists a bijection $\theta : \{1, \ldots, |H^i|\} \to \{1, \ldots, |S|\}$ satisfying the conditions of Definition 4. Since θ preserves the per-thread order of actions and $\mathsf{history}(\tau_1) = S_1$, we can assume that θ maps actions in $\mathsf{history}(\tau_1)$ to the corresponding actions in S_1. Hence, for some traces τ_3 and τ_4, we have

$$\mathsf{history}(\psi) = \psi, \qquad \tau_2 = \tau_3\,\psi\,\tau_4, \qquad \tau^i = \tau_1\,\tau_2 = \tau_1\,\tau_3\,\psi\,\tau_4,$$

and θ maps the ψ in $\mathsf{history}(\tau^i)$ to ψ in S, i.e., $\theta(|\mathsf{history}(\tau_1\,\tau_3)| + 1) = |S_1| + 1$; see Figure 3(a).

Let $\psi = (t, \_)$. We note that, since θ preserves the per-thread order of actions and $\mathsf{history}(\tau_1) = S_1$, we have $\mathsf{history}(\tau_3|_t) = \varepsilon$. We proceed by case analysis on the kind of ψ.

Case I: $\psi \neq (t, \mathsf{txbegin})$. Let $\tau^{i+1} = \tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4$ and $H^{i+1} = \mathsf{history}(\tau^{i+1})$; see Figure 3(b). Intuitively, τ^{i+1} is obtained from $\tau^i = \tau_1\,\tau_3\,\psi\,\tau_4$ by moving all the actions in τ_3 performed by thread t, together with ψ, to the position right after τ_1.

Since $\mathsf{history}(\tau_3|_t) = \varepsilon$, $\mathsf{history}(\tau_1) = S_1$ and $|S_1| = i$, we get:

$$\begin{array}{rcl} H^{i+1}\downarrow_{i+1} & = & (\mathsf{history}(\tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4))\downarrow_{i+1} \\ & = & S_1\,\psi = S\downarrow_{i+1}, \end{array}$$

as required. We also have:

$$\begin{array}{rcl} \tau^{i+1}|_t & = & (\tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4)|_t \\ & = & (\tau_1|_t)\,(\tau_3|_t)\,\psi\,(\tau_4|_t) \\ & = & (\tau_1\,\tau_3\,\psi\,\tau_4)|_t \\ & = & \tau^i|_t; \end{array}$$

$$\begin{array}{rcl} \tau^{i+1}|_{\neg t} & = & (\tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4)|_{\neg t} \\ & = & (\tau_1|_{\neg t})\,(\tau_3|_{\neg t})\,(\tau_4|_{\neg t}) \\ & = & (\tau_1\,\tau_3\,\psi\,\tau_4)|_{\neg t} \\ & = & \tau^i|_{\neg t}. \end{array}$$

Hence, for any thread t', we have $\tau^i|_{t'} = \tau^{i+1}|_{t'}$ and $H^{i+1}|_{t'} = H^i|_{t'} = S|_{t'}$. Any real-time order between two actions that exists in H^{i+1}, but not in H^i, also exists between the actions in S corresponding to them according to θ. Thus, $H^{i+1} \sqsubseteq S$.

Since $\psi \neq (t, \mathsf{txbegin})$ and $\mathsf{history}(\tau_3|_t) = \varepsilon$, all the actions performed by t in the subtrace τ_3 of τ^i are transactional. Hence,

$$\begin{array}{rcl} \mathsf{nontrans}(\tau^{i+1}) & = & \mathsf{nontrans}(\tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4) \\ & = & \mathsf{nontrans}(\tau_1\,\tau_3\,\psi\,\tau_4) \\ & = & \mathsf{nontrans}(\tau^i) \end{array}$$

and therefore $\tau^i \sim \tau^{i+1}$.

Case II: $\psi = (t, \mathsf{txbegin})$. Note that the subtrace τ_3 of τ^i does not contain any committed or aborted actions ψ'. Indeed, assume otherwise. Since $\mathsf{history}(\tau_1) = S_1$, the action in S corresponding to ψ' according to θ would be in S_2. This would mean that the real-time order between ψ' and ψ in H^i is not preserved in S, contradicting our assumption that $H^i \sqsubseteq S$. Thus, for any thread $t' \neq t$, $\tau_3|_{t'}$ consists of some number of non-transactional actions followed by some number of transactional ones, and $\tau_3|_t$ does not contain any transactional actions. Motivated by these observations, we let

$$\tau^{i+1} = \tau_1\,\mathsf{nontrans}(\tau_3)\,\psi\,\mathsf{trans}(\tau_3)\,\tau_4$$

and $H^{i+1} = \mathsf{history}(\tau^{i+1})$; see Figure 3(c).[3] Intuitively, τ^{i+1} is obtained from $\tau^i = \tau_1\,\tau_3\,\psi\,\tau_4$ by moving all transactional actions in τ_3 to the position right before τ_4.

Since $\mathsf{history}(\mathsf{nontrans}(\tau_3)) = \varepsilon$, $\mathsf{history}(\tau_1) = S_1$ and $|S_1| = i$, we get:

$$\begin{array}{rcl} H^{i+1}\downarrow_{i+1} & = & (\mathsf{history}(\tau_1\,\mathsf{nontrans}(\tau_3)\,\psi\,\mathsf{trans}(\tau_3)\,\tau_4)\downarrow_{i+1} \\ & = & S_1\,\psi = S\downarrow_{i+1}, \end{array}$$

as required.

Since for every thread $t' \neq t$, $\tau_3|_{t'}$ consists of non-transactional actions followed by transactional ones,

$$(\mathsf{nontrans}(\tau_3)\,\psi\,\mathsf{trans}(\tau_3))|_{t'} = (\tau_3\,\psi)|_{t'}.$$

Since $\tau_3|_t$ does not contain any transactional actions, $\tau_3 = \mathsf{nontrans}(\tau_3)$ and, hence,

$$(\mathsf{nontrans}(\tau_3)\,\psi\,\mathsf{trans}(\tau_3))|_t = (\tau_3\,\psi)|_t.$$

Thus, for any t'' we have

$$\begin{array}{rcl} \tau^{i+1}|_{t''} & = & (\tau_1\,\mathsf{nontrans}(\tau_3)\,\psi\,\mathsf{trans}(\tau_3)\,\tau_4)|_{t''} \\ & = & (\tau_1\,\tau_3\,\psi\,\tau_4)|_{t''} = \tau^i|_{t''} \end{array}$$

[3] Here we are applying nontrans to a part τ_3 of a well-formed trace τ^i. Note that, in the absence of txbegin actions in τ_3, the definition of nontrans given in Section 4 considers all actions in τ_3 non-transactional.

and $H^{i+1}|_{t''} = H^i|_{t''} = S|_{t''}$. Any real-time order between actions in H^{i+1} also exists in H^i, and hence, $H^{i+1} \sqsubseteq S$. Finally,

$$\begin{array}{rcl} \mathsf{nontrans}(\tau^{i+1}) & = & \mathsf{nontrans}(\tau_1\, \mathsf{nontrans}(\tau_3)\, \psi\, \mathsf{trans}(\tau_3)\, \tau_4) \\ & = & \mathsf{nontrans}(\tau_1\, \tau_3\, \psi\, \tau_4) \\ & = & \mathsf{nontrans}(\tau^i), \end{array}$$

so that $\tau^i \sim \tau^{i+1}$. $\square$

Proof of Lemma 18

Consider s, τ_H and τ_S such that $\tau_H \sim \tau_S$ and $\mathsf{eval}(s, \tau_H) \neq \emptyset$. We need to show that $\mathsf{eval}(s, \tau_S) \neq \emptyset$. The proof of this fact is similar in its structure to that of Lemma 17. We inductively construct a sequence of well-formed traces τ^i, $i = 0..|\tau_S|$ such that

$$\tau^i\downarrow_i = \tau_S\downarrow_i; \quad \tau^i \sim \tau_S; \quad \mathsf{eval}(s, \tau^i) = \mathsf{eval}(s, \tau_H) \neq \emptyset. \quad (6)$$

Then for $i = |\tau_S|$ we get $\tau^i = \tau_S$, which implies the required.

To construct the sequence of traces τ^i, we let $\tau^0 = \tau_H$, so that all the requirements in (6) hold trivially. Assume now that a trace τ^i satisfying (6) has been constructed. Let $\tau_S = \tau_1\, \varphi\, \tau_2$, where $|\tau_1| = i$, and $\varphi = (t, \_)$. By assumption, $\tau^i\downarrow_i = \tau_S\downarrow_i$ and, since $\tau^i \sim \tau_S$, we also have $\tau^i|_t = \tau_S|_t$. Hence, for some traces τ_2' and τ_2'', we get

$$\tau^i = \tau_1\, \tau_2'\, \varphi\, \tau_2'',$$

where τ_2' does not contain any actions by thread t. Let

$$\tau^{i+1} = \tau_1\, \varphi\, \tau_2'\, \tau_2'';$$

then $\tau^{i+1}\downarrow_{i+1} = \tau_S\downarrow_{i+1}$. We now show that $\tau^{i+1} \sim \tau^i$ and $\mathsf{eval}(s, \tau^{i+1}) = \mathsf{eval}(s, \tau^i)$, which implies the required.

First note that $\tau^i|_{t'} = \tau^{i+1}|_{t'}$ for any thread t' since τ_2' does not contain any actions by thread t. Next, consider the case when the action φ in τ^i is non-transactional. Since $\tau^i|_t = \tau_S|_t$, so is the corresponding action φ in τ_S. Assume that some action $\varphi' = (t', \_)$ in the subtrace τ_2' of τ^i is non-transactional as well, where $t' \neq t$. Let φ' be the j-th action by thread t' in τ^i. Since $\tau^i|_{t'} = \tau_S|_{t'}$, φ' is also the j-th action by thread t' in τ_S and is non-transactional in this trace. But then φ' has to be in the subtrace τ_2 of τ_S and thus follow φ, contradicting $\mathsf{nontrans}(\tau^i) = \mathsf{nontrans}(\tau_S)$. Hence, if the action φ in τ^i is non-transactional, then the subtrace τ_2' of τ^i does not contain any non-transactional actions. This implies $\mathsf{nontrans}(\tau^{i+1}) = \mathsf{nontrans}(\tau^i)$ and thus $\tau^{i+1} \sim \tau^i$.

The restrictions on accesses to variables by commands from LPcomm_t and GPcomm_t (stated in Section 2 and formalized in Section 4) imply

PROPOSITION 23. *Assume φ_1 and φ_2 are actions by different threads and, if $\varphi_1 = (t_1, c_1)$ and $\varphi_2 = (t_2, c_2)$, then*

$$(c_1 \in \mathsf{LPcomm}_{t_1} \wedge c_2 \in \mathsf{LPcomm}_{t_2} \uplus \mathsf{GPcomm}_{t_2}) \vee (c_1 \in \mathsf{LPcomm}_{t_1} \uplus \mathsf{GPcomm}_{t_1} \wedge c_2 \in \mathsf{LPcomm}_{t_2}).$$

Then $\mathsf{eval}(s, \varphi_1\, \varphi_2) = \mathsf{eval}(s, \varphi_2\, \varphi_1)$ for any state s.

Since τ^i is well-formed, by Definition 12 for any action (t', c) in it we have that $c \in \mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t$ and, if $c \in \mathsf{GPcomm}_t$, then the action is non-transactional. Given this and the properties of τ_2' established above, by applying Proposition 23 repeatedly we get that $\mathsf{eval}(s', \varphi\, \tau_2') = \mathsf{eval}(s', \tau_2'\, \varphi)$ for any state s'. Hence,

$$\mathsf{eval}(s, \tau^{i+1}) = \mathsf{eval}(s, \tau_1\, \varphi\, \tau_2'\, \tau_2'') = \mathsf{eval}(s, \tau_1\, \tau_2'\, \varphi\, \tau_2'') = \mathsf{eval}(s, \tau^i). \quad \square$$

7. NECESSITY OF THE OPACITY RELATION

In this section we prove that observational refinement with respect to states implies the opacity relation restricted to the complete subsets of the transactional systems. We only give a proof sketch here and defer the full proof to [2, Appendix B].

THEOREM 24. $\mathcal{H}_C \preceq_{\mathsf{state}} \mathcal{H}_A \implies \mathcal{H}_C|_{\mathsf{complete}} \sqsubseteq \mathcal{H}_A|_{\mathsf{complete}}$.

PROOF SKETCH. For every complete history $H \in \mathcal{H}_C$ we construct a program P_H where every thread performs the sequence of transactions specified by H, records the return values obtained from methods of transactional objects, and monitors whether the real-time order between actions includes that in H.

In more detail, given that the history H is well-formed, we construct the code of every thread t as a sequence of `atomic` blocks, corresponding to txbegin, committed and aborted actions in $H|_t$, which perform the sequence of method invocations determined by call and ret actions in $H|_t$. We record the return value for every method invocation in a dedicated variable local to the corresponding thread, which is never rewritten (this is possible because H is finite). The return status of every transaction is also recorded in a dedicated local variable. As a result, from the final state of an execution of P_H, we can reconstruct the interaction of each thread with the transactional system. (We rely here on the fact that the histories considered are complete; our notion of observation is not strong enough to distinguish between, e.g., histories $H\, \varphi\, H'$ and $H\, H'$ where φ is a call action that does not return.)

To check whether an execution of P_H complies with the real-time order in H, we exploit the ability of threads to communicate via global variables outside transactions. For each transaction in H, we introduce a global variable g, which is initially 0 and is set to 1 by the thread executing the transaction right after the transaction completes, by a command following the corresponding `atomic` block. Before starting a transaction, each thread checks whether all transactions preceding this one in the real-time order in H have finished by reading the corresponding g variables. The outcome is recorded in a dedicated local variable.

Let s be a state with all variables set to distinguished initial values. From the above it follows that, given any trace $\tau \in [\![P_H]\!](s)$, by observing the states in $\mathsf{eval}(s, \tau)$, we can unambiguously determine whether the per-thread projections of $\mathsf{history}(\tau)$ are equal to those of H and whether the real-time order in $\mathsf{history}(\tau)$ includes that in H.

Now given $H \in \mathcal{H}_C$, we construct a trace $\tau \in [\![P_H]\!](s, \mathcal{H}_C)$ such that $\mathsf{history}(\tau) = H$ and $\mathsf{eval}(s, \tau) = \{s'\}$, where the state s' signals the above correspondence of the trace to H. By Definition 14, there exists a trace $\tau' \in [\![P_H]\!](s, \mathcal{H}_A)$ such that $\mathsf{eval}(s, \tau') = \{s'\}$. But then the per-thread projections of $S = \mathsf{history}(\tau')$ are equal to those of H and the real-time order in H is preserved in S. By Definition 4, this implies $H \sqsubseteq S$. $\square$

The proof highlights the features of our programming model that lead to the necessity result. First, the ability to access global variables outside transactions allows us to monitor the real-time order. Second, we use the ability of programs to observe values assigned to local variables during the execution of a transaction, even if the transaction aborts: P_H records the return values of method invocations by aborted transactions in special local variables, which can then be observed in final states. This necessitates treating aborted transactions in the same way as committed ones in the consistency definition.

8. RELATED WORK AND DISCUSSION

Previous work has studied transactional memory consistency by:

- investigating the semantics of different programming languages with atomic blocks and the feasibility of their efficient implementation [1, 11, 19]; or
- defining consistency conditions for TM [3,5,8,9,18] and proving that particular TM implementations validate them [4, 9].

Thus, previous work has tended to address the issue from the perspective of either programming languages or TM implementations and has not tried to relate these two levels in a formal manner. An exception is the work by Harris et al. [12], which proved that a specific TM implementation, Bartok-STM, validates a particular semantics of atomic blocks in a programming language.

This paper tries to fill in the gap in existing studies by relating the semantics of a programming language with atomic blocks to that of a TM system implementing them. Our work is complementary to previous proofs that certain TM systems satisfy opacity [4], as it lifts such results to the language level. Our work is also more general than that of Harris et al. [12], since our results allow establishing observational refinement for *any* TM implementation satisfying opacity. However, some of the above-mentioned papers [1, 19] investigated advanced language interfaces that we do not consider, such as nested transactions and access to shared data both inside and outside transactions.

This paper employs a well-known technique from the theory of programming languages, observational refinement [13, 14], to explore the most appropriate way to specify TM consistency. Observational refinement has previously been used to characterize correctness criteria for libraries of concurrent data structures. Thus, Filipovic et al. [6] proved that, in this setting sequential consistency [17] is necessary and sufficient for observational refinement, and so is linearizability [15] when client programs can interact via shared global variables. Gotsman and Yang [7] adjusted linearizability to account for infinite computations and showed its sufficiency for observational refinement in the case when the client can observe the validity of liveness properties. Our work takes this approach from the simpler setting of concurrent libraries to the more elaborate setup of transactional memory. Since we allow the abstract transactional system to have incomplete transactions, we hope that in the future we can generalize our consistency condition to specify liveness properties, along the lines of [7].

Opacity requires the TM behavior observed by aborted transactions to be as consistent as that observed by committed ones. Other proposed consistency conditions tried to relax this requirement. For example, *virtual world consistency (VWC)* [16] requires the behavior observed by an individual aborted transaction to be consistent only with the committed transactions from which it reads and those previously committed by the same thread. Our necessity result implies that VWC does not imply observational refinement for our programming language. However, this does not rule out the viability of VWC and related notions as a consistency condition for TM. VWC may well imply observational refinement for a programming language in which aborted transactions do not affect the rest of the computation (in particular, their modifications to local variables are rolled back) and a weaker notion of observations. This paper establishes an approach for evaluating and comparing TM consistency conditions, and in future work, we hope to apply it to VWC and other conditions, such as TMS [5, 18] and DU-opacity [3]. We also intend to investigate the behaviour of incomplete histories with respect to observational refinement, whose treatment by the opacity relation causes our result to fall short of the strict equivalence between the relation and the observational refinement.

Acknowledgements

This work is supported by EU FP7 projects TRANSFORM (238639) and ADVENT (308830). We thank Victor Luchangco, Eran Yahav, Hongseok Yang and the reviewers for helpful comments.

9. REFERENCES

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, pages 63–74, 2008.

[2] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. Technical Report CS-2013-04, Department of Computer Science, Technion, 2013.

[3] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety of deferred update in transactional memory. In *ICDCS*, 2013. To appear.

[4] A. Bieniusa and P. Thiemann. Proving isolation properties for software transactional memory. In *ESOP*, pages 38–56, 2011.

[5] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, pages 1–31, March 2012.

[6] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *ESOP*, pages 252–266, 2009.

[7] A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP (2)*, pages 453–465, 2011.

[8] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.

[9] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2011.

[10] T. Harris, J. Larus, and R. Rajwar. *Transactional memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[11] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. In *PPOPP*, pages 48–60, 2005.

[12] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI*, pages 14–25, 2006.

[13] J. He, C. Hoare, and J. Sanders. Data refinement refined. In *ESOP*, pages 187–196, 1986.

[14] J. He, C. Hoare, and J. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71 – 76, 1987.

[15] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[16] D. Imbs, J. R. G. de Mendívil, and M. Raynal. Brief announcement: virtual world consistency: a new condition for STM systems. In *PODC*, pages 280–281, 2009.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[18] M. Lesani, V. Luchangco, and M. Moir. Putting opacity in its place. In *WTTM*, 2012.

[19] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL*, pages 51–62, 2008.