

Security Correctness for Secure Nested Transactions

Dominic Duggan
Stevens Institute of Technology
dduggan@stevens.edu

Ye Wu
Stevens Institute of Technology
ywu1@cs.stevens.edu

Stevens CS Technical Report 2011-4, July 2011

Abstract

This article considers the synthesis of two long-standing lines of research in computer security: security correctness for multilevel databases, and language-based security. The motivation is an approach to supporting multilevel and multilateral security for a wide class of enterprise applications, those of concurrent transactional applications. The approach extends nested transactions with *retroactive abort*, a new form of semantics for transactional execution, motivated by security concerns. A semantics is given in terms of a local constrained labelled transition system, the **Tau**_{One} calculus. This allows a noninterference result to be verified based on adapting results on observational equivalence from concurrency theory.

1 Introduction

Computer security is concerned with ensuring three properties of information assets: confidentiality, integrity and availability. Giving precise definitions of these properties, and how to achieve them, has been a preoccupation of computer security researchers now for several decades. Noninterference has provided gainful employment for security researchers for the last quarter of a century, and is still something of a Gold Standard for end-to-end security in multilevel systems. For example, with increasing emphasis on accountability for secure systems, noninterference can establish a baseline for authorized release of sensitive information, based on clearly delineated declassification events recorded in an audit log.

In the realm of multilevel databases, confidentiality was defined in terms of noninterference for transactional execution at least twenty years ago. Consider for example the following program, that contains a high transaction T_1 and a low transaction T_2 :

```
intLow X, Y, Z;  
 $T_1^{\text{High}}$ : lock(X); while (1) ;  
 $T_2^{\text{Low}}$ : (lock(X); Z=0;) || (lock(Y); Z=1)
```

Preventing the writing of sensitive information to a “low” database variable is insufficient, since the use of locks to synchronize accesses to the database provide a covert channel. In this example, T_1 signals to T_2 by locking X but not Y. In multilevel databases, this leak is prevented by allowing the low transaction to implicitly pre-empt the high transaction when the latter holds a resource that former requires [3].

Volpano and Smith [39] noted that the certification of multilevel secrecy in programs [12, 13] could be formulated as a type system, and this idea has been adopted by many researchers [38, 27, 34, 4]. The key

insight in this work is that noninterference can be related to the control flow in a program, so that indirect leaks through the control flow may be prevented via a type-based control flow analysis. For example, in the following program, there is an apparent information leak due to the writing to “low” variable Y after reading “high” variable X:

```
intHigh X; intLow Y;  
if (X==0) Y=0; else Y=1;
```

The fact that 0 or 1 is written to the “low” variable Y depends on whether the value of the “high” variable X is 0 or 1. The leak is prevented by the type system, which only allows writes to “high” variables in a local context where “high” variables have been tested. This area has been extensively studied in the language-based security community, and some of this research has informed research in multilevel secure operating systems [14, 41].

Ensuring noninterference in concurrent and distributed systems is still a challenge. For example, *termination leaks* in multi-threaded systems allow simulation of the leak prevented by the type system in the sequential example above:

```
intHigh X, Y; intLow Z;  
(X=0; Y=1;)  
|| (while (X==0); Z=0;) || (while (Y==0); Z=1;)
```

Extensions of earlier type systems to prevent such information leaks in multithreaded environments have been proposed, but these extensions miss the point: busy-waiting is one form of synchronization between threads, and it is not clear how to prevent information leaks once synchronization is allowed between threads of different security levels.

Why should threads or processes at different security levels need to synchronize? Obvious examples can be found in the realm of enterprise data processing systems, where the transaction processing monitor CICS alone accounts for twenty billion financial transactions **a day**. High and low transactions accessing a shared database need to synchronize so that high processes only see consistent views of the database. This has been supported for almost twenty years by multilevel databases. On the other hand, the latter require that each transaction execute either always at the high level or always at the low level. The notion of low programs being limited to “high” writes after testing “high” variables is not available.

Most of the work in the semantics of information flow control for concurrent processes has been done in the realm of process algebras. These approaches at the least prevent control flow dependencies from high to low processes, and may go further to prevent more indirect timing channel leaks based on the non-deterministic execution of processes. Various approaches have been proposed to weaken this restriction, for pragmatic reasons. For example, the approach of BNDC [16, 17, 18] allows high actions to be ignored, provided that any low actions that are dependent on those high actions can be simulated by low actions alone. However it is not clear what the pragmatic interpretation of this is, if the point of allowing causal dependencies from high to low actions is to allow synchronization between high and low processes. The “noise” that masks the communications of high processes also precludes meaningful synchronization between the two. Another approach is to use linear types [24] to constrain the use of synchronization primitives such as locks: a process with a linearly-typed lock variable must use that lock exactly once. This makes the usage of shared resources completely deterministic and removes the potential information leak due to, for example, blocking a low process by never releasing a lock. However linear types are difficult to explain and check, and are unfamiliar to programmers.

Nested transactions were introduced [26] to provide a transactional underpinning to remote procedure call chains. Nested transactions allow a parent transaction to commit despite the failure of child transactions,

allowing a form of programming based on recovering from the failure of remote services. Nested transactions are supported in for example SQL Server, but are not part of the Java Enterprise Edition transaction model. In the latter, annotations may be used to specify that a bean starts a new transaction, independent of any parent transaction, when a service method is called. However if the parent transaction aborts, this results in a committed child transaction that has become an orphan. If the child transaction inherits the transaction of the parent, then failure of the latter forces abort of the former. This prevents low transactions from calling into high level transactions, reflecting the state of multilevel databases twenty years ago. On the other hand, nested transactions are part of the coordination model for the WS-* Web services stack that underpins business process modeling frameworks such as BPEL and BPMN.

This article proposes a model for allowing high and low processes to coordinate their activities in multi-level secure systems, without leaking information. This model is based on extending the nested transaction model to support transactional coordination between high and low transactions, while ensuring a noninterference property for their interactions. The semantics of nested transactions is extended with *retroactive abort* in order to support this extension.

Sect. 2 provides motivation for *retroactive abort*. Sect. 3 provides an informal introduction to **Tau_{One}**, a formal language for describing the semantics of nested transactions with retroactive abort. Sect. 4 presents the semantics of **Tau_{One}**. A security type system for nested transactions is considered in Sect. 5. We verify noninterference for this nested transactional calculus in Sect. 6. We consider related work in Sect. 7 and conclude in Sect. 8.

2 Motivation

In this section we provide some motivation for the challenges with incorporating information flow into transactional semantics. We discuss these issues in the realm of lock-based concurrency control, since that is the form of concurrency control for which the semantics of nested transactions is defined. Extending this approach to timestamp-based and optimistic concurrency control is a topic for future work.

The problem with termination leaks, as demonstrated in the previous section, is that a high process, potentially a “Trojan horse,” may choose to signal to low-level processes by enabling the execution of one of two low processes. It does this by having two high pieces of code executing infinite loops, with low continuations. The signaling process sets a variable to make on these loop guards false, but not the other, thereby releasing one of the two threads. Once the high loop terminates, the subsequent low code can execute and write to low variables. This is symptomatic of synchronization between high and low processes. For example, an infinite loop may be used to for busy-waiting synchronization. If we abstract from the implementation of synchronization primitives such as mutex locks, the issues just discussed now arise at the level of lock acquisition and release.

Assume that transactions use locks for concurrency control. It must be possible for “high” transactions to acquire locks on low variables, in order to synchronize their reads of those variables with low process updates. The danger is that a high transaction will use locks to control the continuation of low processes so as to signal to those low processes. For example, a high transaction will acquire locks on low variables X and Y. Although it cannot write to those variables, it can release the lock on one, say X, thereby releasing a low process that is trying to acquire the X lock, while retaining the lock on Y and preventing a low process from proceeding that is attempting to acquire the Y lock. Other low processes may observe that X has proceeded while Y is locked, demonstrating a covert channel from high to low processes. To prevent this leak, we require that a high process must abort if it has a lock on a low variable that a low process is trying to access. This makes the high process vulnerable to starvation due to repeated aborts caused by low processes

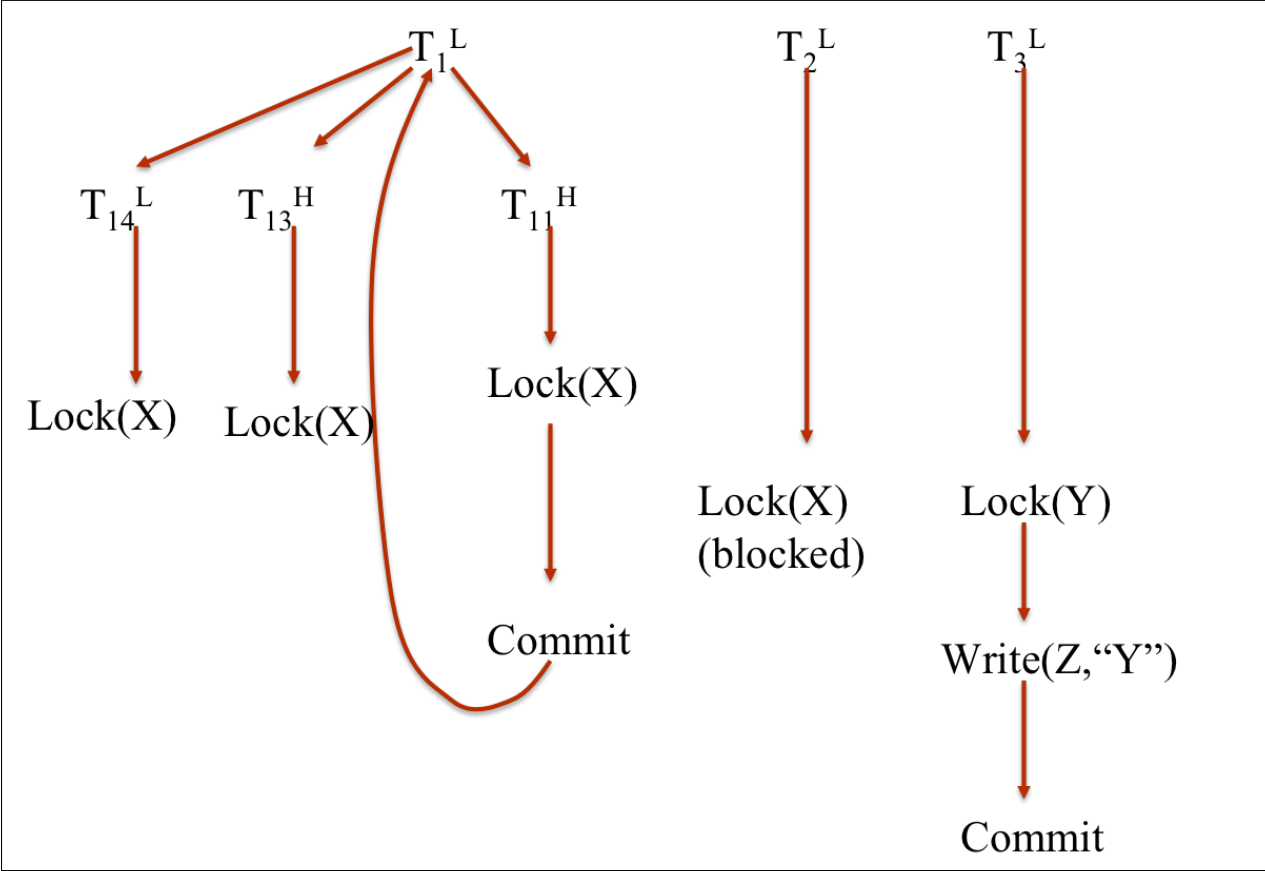


Figure 1: Information Leak with High Child Transaction

trying to acquire the same low variables. We do not consider starvation further and focus on the causal leak due to holding of locks. In a practical setting, starvation of high processes may be detected through careful monitoring of the system’s execution.

Although low transactions must necessarily be able to pre-emptively abort high transactions, it is equally important that high transactions not be able to abort low transactions. We must assume that the abort of low transactions is observable by low processes. This has implications for the design of the transaction system. Assume that threads and transactions are orthogonal, so that a thread may spawn new threads within the transaction within which it is currently executing. Let the “level” of a thread reflect the level of the variables that it has examined in its context. If a thread has tested the value of a high variable, then its level must be high, preventing it from writing to low-level variables. If we have a transaction that contains both low and high threads, and a high thread acquires a lock, then a low thread (outside that transaction) acquiring that lock can implicitly abort the transaction, and in the process abort any low threads within that transaction. Therefore we cannot have low and high threads within the same transaction. Therefore we refer to transactions as being high or low, reflecting that all threads within a transaction must have the same level. This demonstrates that in the flat transactional model, there can be no intermixing of low and high computations in the single transaction, as in the case of scenarios considered for type systems inspired by Volpano and Smith. This is part of our motivation for considering nested transactions.

In some situations, it may be desirable to have high and low threads to collaborate in a transaction. An example is a low thread that needs to authenticate in order to perform a write, but the credentials are high data. We can allow this cooperation, within some limits, if we adopt the semantics of *nested transactions* [26]. Nested transactions allow trees of transactions, intuitively reflecting call trees in systems of remote procedure calls. The root of this tree corresponds to the first procedure call, and in general a child node in the tree reflects procedure calls that arose from the execution of the parent node procedure call. One of the interesting aspects of this semantics is that, for all transactions but the root transaction in such a tree, commit is a tentative operation. Even if a child transaction commits, its parent may choose to abort, and that in turn requires the child transaction tentative commit to be undone. On the other hand, abort of a child transaction does not require the parent transaction to be aborted.

Therefore we can allow high and low threads to co-exist safely in a transaction, provided we isolate them in subtransactions according to their level. Furthermore, although we allow low transactions as parents of high transactions, we disallow the inverse: a high transaction cannot be the parent of a child transaction, for the obvious reason that the former may exploit a covert channel by causing the abort of the latter.

Fig. 1 illustrates the challenges that may arise. In this example, the low transactions T_1^L , T_2^L and T_3^L are cooperating with the high transaction $T_{1.1}^H$ in order to create a covert channel that bypasses the level restrictions on information flow. $T_{1.1}^H$ is a child of T_1^L . The high child transaction $T_{1.1}^H$ acquires the lock for the variable X, in order to establish a covert channel to a low transaction. This high transaction commits, releasing the lock on the variable to its parent (since its commitment must be tentative). The low transactions T_2^L and T_3^L attempt to acquire locks on variables X and Y, respectively. T_3^L acquires the lock on Y, prints a message to this effect, and commits. Since it is outside of any other transaction, its effects are now publicly visible. On the other hand, T_2^L is blocked on attempting to lock X, which was originally locked by $T_{1.1}^H$. Were the latter still active, it would be forced to abort by T_2^L and the lock released. However the lock is now held by the parent of the high transaction, T_1^L , even if this low parent is unaware of the lock it has acquired via the actions of its child. To fix this problem, we require that the high child transaction $T_{1.1}^H$ of T_1^L be *retroactively* aborted. This is possible because the effects of any successful transactions cannot be made visible outside a nested transaction until the root transaction succeeds, and the low parent of a high transaction obviously cannot be aware of whether its high child aborted or committed.

Fig. 1 illustrates two other scenarios related to this. Suppose another high child transaction of $T_{1,3}^H$ has inherited the lock (from its parent T_1^L) that was originally acquired by $T_{1,1}^H$. In this case, $T_{1,3}^H$ will be aborted when the low transaction T_2^L attempts to lock X. Suppose on the other hand that the low child $T_{1,4}^L$ of T_1^L has acquired the lock on X that was originally acquired by $T_{1,1}^H$. If we allow T_2^L to pre-emptively abort $T_{1,4}^L$, then this is caused indirectly by $T_{1,1}^H$, so this is a form of abort dependency from high to low transactions that we are claiming to avoid. In this case, we can say that the low transaction $T_{1,4}^L$ is oblivious of the fact that the lock has been acquired due to inheritance and anti-inheritance from a high sibling, and this amounts to a scenario where a low transaction is blocked due to a resource being held by another low transaction. In this case, there is no information leak and the participation of high transactions in the overall computation is unknown to the low transaction.

Fig. 2 and Fig. 3 contain some example scenarios regarding the release of locks held by high child transactions that are requested by external low transactions, where the root of the current transaction is low. In Fig. 2(a), the high child $T_{1,1}^H$ has acquired a lock on X, that T_2^L is requesting. To avoid a termination leak where T_2^L may be made to wait indefinitely, we abort $T_{1,1}^H$ and release the lock on X to T_2^L . This is safe because the lock was initially acquired by $T_{1,1}^H$, so no part of the root transaction T_1^L outside of T_2^L has seen any changes to X.

In Fig. 2(b), the lock on X was acquired by the low child $T_{1,1}^L$ and then inherited by the high descendant $T_{1,1,1}^H$. As before, the high transaction $T_{1,1,1}^H$ holding the lock on X must be forced to abort. However we cannot release the lock to T_2^L because $T_{1,1}^L$ still expects to hold the lock that it acquired earlier. So we must release the lock to the low parent $T_{1,1}^L$ when we abort the high child $T_{1,1,1}^H$. At this point, the low transaction T_2^L is blocked waiting on a lock to be released that is held by another low transaction $T_{1,1}^L$, so at least we do not have a termination leak controlled by a high transaction.

Fig. 3 illustrates a scenario where a child transaction $T_{1,1}^?$ has acquired the lock on X, committed and released that lock back to the parent T_1^L (the latter is said to have “anti-inherited” the lock). The lock has then been inherited by the high child transaction $T_{1,2}^H$. What happens when the outside transaction T_2^L requests the lock depends on the level of $T_{1,1}^?$. If the latter is high, then it must be retroactively aborted along with the high transaction $T_{1,2}^H$, and the lock globally released to T_2^L . This is safe to do because any updates by $T_{1,1}^H$ will be to high variables, so T_1^L will be unaffected if we retroactively undo the changes done by $T_{1,1}^H$. If the level of $T_{1,1}^?$ is low, on the other hand, then the lock was originally acquired by a low sibling and released back to the low parent, so we must retain the lock in the low parent when we abort $T_{1,2}^H$.

3 Logs for Transactional State

In this section, we provide the syntax of our transactional language, and consider the most salient innovation in this language, the use of logs to record transaction state. We refer to the calculus of secure nested transactions presented here as **Tau_{One}**.

In language-based security, there are two broad approaches to language semantics: a shared-memory computational model, with a conventional imperative language, or a process calculus based on processes that exchange messages. The latter has the advantage of a rich theory of observational equivalence, that can be used to reason about noninterference based on observable behavior. This is particularly important when reasoning about the semantics of computations that may not terminate. The area of applications that we are interested in are transactional enterprise applications, communicated via message-passing. Process calculi are the best match with the semantics of applications in this domain.

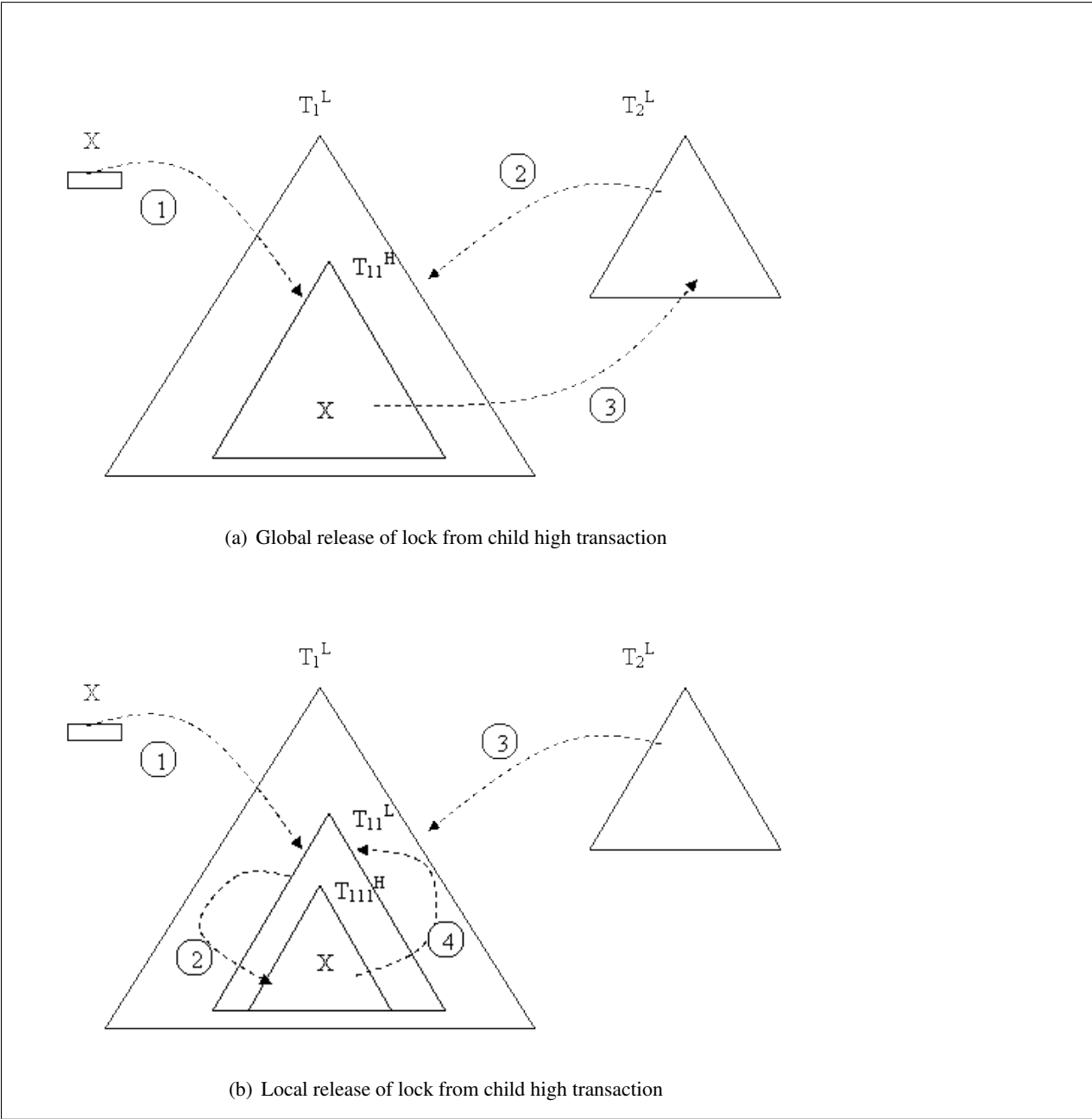


Figure 2: Releasing locks held by child transactions

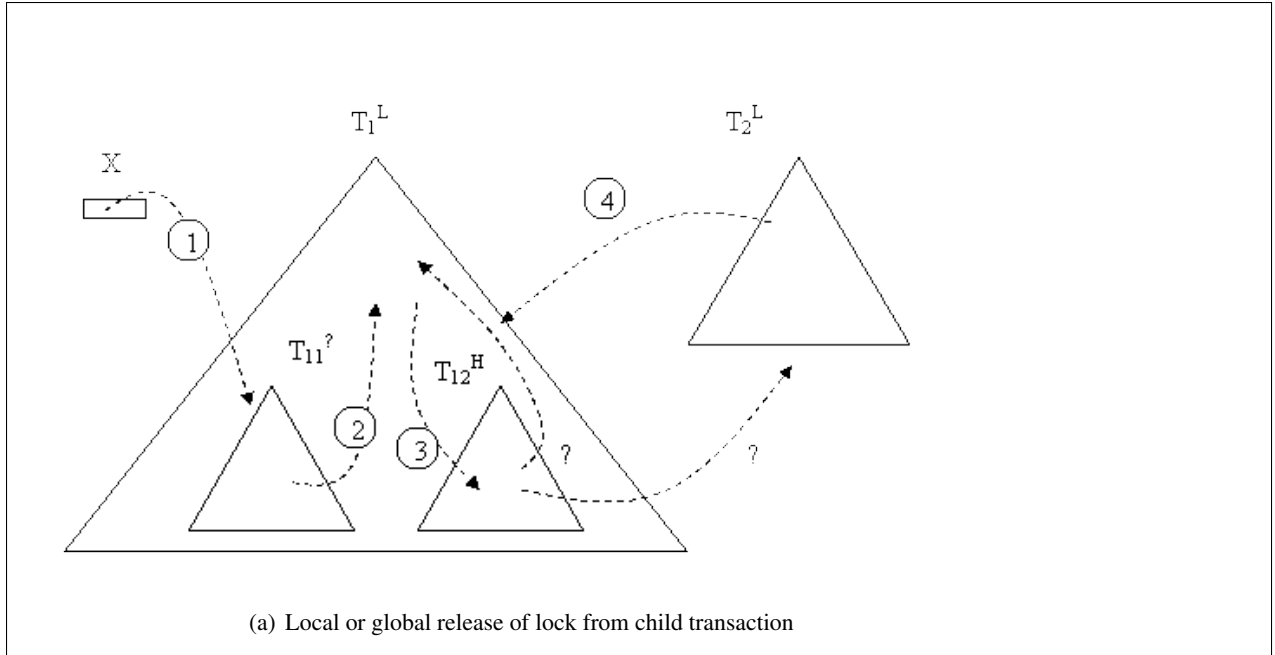


Figure 3: Releasing locks held by child transactions (cont'd)

We base our language on the asynchronous pi-calculus. This calculus must support communication paths from low to high processes, so the latter can receive data from the former. It might be assumed that with asynchronous message-passing, it is safe to allow low processes to “write up” by sending low security level messages to high processes. However as noted by Pottier [29], it is then possible for high processes to signal indirectly to low processes by consuming messages of low security level. There are alternatives to observational equivalence for process equality, that can allow messages to be transmitted asynchronously from low to high processes. For example the may-testing equivalence used by Hennessy and Riely [21] accomodates such communication. However may-testing equivalence is a weak notion of process equivalence that equates to trace equivalence, and does not consider the deadlock behavior of processes. We choose observational equivalence to avoid controversy.

Synchronization in message-passing calculi is usually implemented using message-passing. Because of the aforesaid issues with traffic analysis in message-passing, we add a special form of message, that we refer to as “locks¹.” Our notion of locks serves two purposes:

1. It represents the most familiar form of synchronization in distributed transactions to the lay reader, the one that is used in the description of nested transactions in every textbook on distributed systems. Ph.D. theses have been awarded on semantics for nested transactions with other forms of synchronization. We do not at this time attempt to incorporate other forms of concurrency control semantics, such as timestamp-based concurrency control or optimistic concurrency control. We rely instead on providing a semantics based on the most widely understood synchronization mechanism for nested transactions.

¹Even if lock-based concurrency control is replaced by some other notion, such as optimistic concurrency control, we will still require messages from low to high processes to be handled in a special *linear* fashion. Therefore we will still need a construct similar to these messages, whether we call them locks or something.

2. As we have seen above, with a suitably refined notion of observational equivalence, we cannot allow “low” messages to be received by “high” processes, since this would allow leaks based on traffic analysis. In database applications, high processes may read low variables without leaking information. However the application domain that we are interested in is transactional enterprise applications, and in this domain processes communicate via message-passing rather than writing to shared global variables. Backend databases are encapsulated by service interfaces. Hence our focus on message-passing for communication, reflecting the use of transactional queues in enterprise applications. “Locks” provide a safe form of communication, based on messages that the runtime ensures are handled linearly. This linearity in message-passing is also the key to ensuring noninterference for secure type systems

The extension of the pi-calculus with logs is intended to separate the mechanics of the language runtime from the conditions that are required for correct transactional execution. These conditions are defined as logical constraints entailed by log entries. This approach avoids defining a system that overspecifies the implementation of retroactive abort. Checking of conditions with respect to the logs in turn can be combined with two-phase commit, as is done in practice for distributed systems.

The syntax of the language is provided in Fig. 4. We assume the following spaces of variables and names:

$$\begin{aligned}
 a, b, c, \dots &\in \text{Channel name} \\
 x, y, z, w &\in \text{Variable} \\
 \vec{t} &\in \text{Transaction id} \\
 k &\in \text{Event id}
 \end{aligned}$$

The only values in the language are channel names, represented by constants a, b, c, \dots . As mentioned, some channels have special significance in their use as locks: they have the property that they are always released by a transaction, whether that transaction commits or aborts. Channels and locks have security levels, as explained in Sect. 6 when we introduce the type system. These security levels stratify channels into high and low channels, with high channels only usable by high processes and similarly for low channels and low processes. Locks are similarly stratified into high and low, but low locks may be acquired by high processes.

We assume the definition of metafunctions $bn(_)$ and $fn(_)$ for computing the set of bound and free names, respectively, in a syntactic term. We also assume the definition of the metafunction $fv(_)$ for computing the set of free variables in a syntactic term.

Each transaction is identified by a sequence of transaction identifiers $\vec{t} = (t_1, \dots, t_k)$ for some k . Here t_1 is intended to be the root transaction, and the complete path identifies a nested transaction and all of its ancestors. We denote the prefix relation between sequences by \leq , so we have:

$$\vec{t}_1 \leq \vec{t}_2 \text{ iff } \vec{t}_2 = \vec{t}_1 . \vec{t}'_1$$

where the period denotes sequence concatenation. Note that $\vec{t}_1 \leq \vec{t}_2$ means that the former transaction is an ancestor of the latter. This is used extensively in what follows.

The semantics of the language needs to track dependencies between transactions. In one dimension of this two-dimensional calculus, the dependency is from parent transactions to child transactions. Failure of the parent transaction forces failure of the child, even if the child has tentatively committed. This is to be consistent with the view that no updates propagate from an aborted transaction, including from any of its child transactions. Therefore a transaction type includes the name of its parent transaction, to record this dependency.

There are richer dependencies in our calculus than parent-child. Even when a high transaction commits and its effects are consumed by other transactions, it is possible for the original high transaction to

$C \in \text{Channel Type}$	$::=$	$(\vec{C})^\ell$	Message channel
		$\text{Lock}(\vec{C})^\ell$	Lock
		Unit^ℓ	Unit
$\ell \in \text{Security Level}$	$::=$	High Low	
$T \in \text{Type}$	$::=$	C	Channel type
		$\text{Trans}(\ell)$	Transaction type
		$\text{Event}(\vec{t}, \ell)$	Event type
$w \in \text{Name}$	$::=$	a \vec{t} k	
$v \in \text{Value}$	$::=$	a x $()$	
$A \in \text{Agent}$	$::=$	$\vec{t}P$	Agent process
		$A_1 A_2$	Composition of agents
		$\llbracket \mathcal{L} \rrbracket$	Local log
		$(va:C)A$	Local name
$P \in \text{Proc}$	$::=$	$\hat{v}\vec{v}$	Send message
		$\check{a}\vec{x}P$	Receive message
		$(v_1=v_2) \rightarrow P_1 \square P_2$	Internal choice
		$P_1 + P_2$	External choice
		$\vec{t}[P]$	Launch transaction
		$P_1 P_2$	Fork process
		$(va:C)P$	New channel
		$\text{repl } P$	Replicate
		stop	Stopped
		\square	Commit or abort
		$\text{await } t[\square] \text{ then } P$	Test status
$\square \in \text{Status}$	$::=$	\square	Commit
		\boxtimes	Abort
$V \in \text{Env}$	$::=$	ε	Empty env
		$(a : C)$	Channel decl
		$(t : \text{Trans}(\ell))$	Transaction decl
		$(k : \text{Event}(\vec{t}, \ell))$	Event declaration
		$V_1.V_2$	Append envs
$\mathcal{L} \in \text{Log}$	$::=$	true	Empty log
		$\mathcal{L}_1 \wedge \mathcal{L}_2$	Join logs
		$k::\vec{t} \hat{a} \vec{c}$	Log send
		$k::\vec{t} \check{a} \vec{c}$	Log receive
		$k_1 \searrow k_2$	Mesg exchanged
		$k \text{ undone}$	Undone receive
		$k_1 \curvearrowright k_2$	Lock transferred
		$\vec{t} \square$	Trans commit
		$\vec{t} \boxtimes$	Trans abort

Figure 4: **Tau**_{One} Syntax

be retroactively aborted and any high successors be subsequently aborted along with it. This is the second dimension of dependencies in this calculus. Therefore we need to track dependencies to propagate cascading aborts of high transactions. Note however that aborts will only cascade within the scope of a parent transaction that contains the high transactions that abort, so the notion of cascading aborts is no worse than is already found with nested transactions. In order to track dependencies as a result of message-passing and synchronization, we use event identifiers k to uniquely identify significant events in the logs.

The syntax of types is provided in Fig. 4. We assume a security type system to prevent information flow leaks, by classifying data as High or Low. The details of this type system are provided in Sect. 6. These security levels ℓ decorate the types of message channels $(\vec{C})^\ell$ and locks $\text{Lock}(\vec{C})^\ell$, and reflect restrictions on information that can be exchanged as a result of synchronization. Transactions are either “high” or “low,” as reflected by their types, and can only have effects (sending and receiving of messages) based on their allowed security level. Whereas in sequential languages, a low thread can raise its security level to high in order to make high side effects, in our language a low transaction must spawn a high child transaction to have high effects. As discussed earlier, if high and low processes occupied the same transaction, a covert channel would be available by having the high process abort the shared transaction.

Both messages received and locks acquired by a transaction are recorded in the logs. If the transaction aborts, these message receive and lock acquisition events are undone, releasing the messages and locks back to their original state. As a transaction executes, messages that are intended to be the output of that transaction are “buffered” by limiting their visibility until the transaction commits. Once the transaction commits, those output messages become visible to the parent transaction, and may be received by processes in that parent transaction or descendants of the parent transaction. If a released message is received by a descendant of the parent transaction (or of some ancestor of the sending transaction), then that message receipt is recorded in the log, and a failure dependency established between the sending and receiving transactions. In the general semantics of transactions, this dependency has no effect since the releasing transaction has tentatively committed. The only way for it to subsequently abort is if an ancestor, such as the parent, aborts. But in the latter case, the receiving transaction will also be forced to abort. Therefore the failure dependency has little import.

These failure dependencies take on more significance when we allow transactions of low security level to retroactively abort transactions of high security level. A high security transaction may have acquired the lock, committed, and released a message. This message was received by a (high) sibling, introducing a failure dependency. The committed high transaction is now retroactively aborted, because it still holds the lock required by a low transaction. This abort forces abort of the other transaction that has become failure dependent upon it. Assuming both high transactions have a low parent, the latter is unaware of the changing status of its children.

Locks acquired by transactions must be treated judiciously once those transactions commit. The locks should be released back to the parent transaction. Therefore logs track the acquisition of locks by a transaction, both for undoing their acquisition if the transaction aborts, and for releasing those locks back to the parent if the transaction commits. Unlike messages, locks do not induce failure dependencies. In particular, if a transaction acquires a lock after it was released by a committed transaction, it does not become failure dependent on retroactive abort of the latter transaction.

The language includes asynchronous message sending and blocking message receive operations. The message send operation $\hat{a} \vec{v}$ outputs values \vec{v} on channel a , where the latter may be sent as part of another message between processes. The message receive operation $\check{a} \vec{x} P$ unpacks a message received on channel a into local variables \vec{x} , and then executes the continuation process P . The accent on the channel names is intended to suggest “upload” and “download” respectively. We enrich these basic message-passing opera-

tions with both internal and external choice operations ($(v_1=v_2) \rightarrow P_1 \square P_2$ and $P_1 + P_2$, respectively), as well as a replication operation $\text{repl } P$. The latter is useful for defining recursive processes. We assume that all processes that are ready to input a message have the form

$$\vec{t} \sum \{\check{a} \vec{x} P\}.$$

In other words, a process may use external choice to select between different input channels. A facility for timeouts could easily be added. A process may launch a new transaction $\vec{t}[P]$ that executes nested within any transaction that encompasses the launching process. The language includes parallel composition and name scoping constructs for each of the forms of constants in the language, as well as a stopped process stop .

The semantics includes logs \mathcal{L} . These hold information both for reasoning about the status of transactions (e.g. committed or aborted), and also for recovering from the abort of a transaction by undoing any visible effects it has had. These take the form of messages consumed or locks acquired during the execution of the transaction. The sending or receiving of a message, and the transfer of a lock, is recorded with a unique event identifier k in the log. The type of this event identifier reflects the transaction in which event occurred. The transaction in turn has an associated security level. The following metafunctions are useful for extracting this metainformation from event and transaction identifiers, where V is a context mapping names and variables to their types:

$$\begin{aligned} \text{tid}(V, k) &= \vec{t} \text{ if } V(k) = \text{Event}(\vec{t}, \ell) \\ \text{lev}(V, \vec{t}) &= \ell \text{ if } \vec{t} = \vec{t}_0.t \text{ and } V(t) = \text{Trans}(\ell) \\ \text{lev}(V, k) &= \ell \text{ if } V(k) = \text{Event}(\vec{t}, \ell) \\ \text{lev}(V, x) &= \text{lev}(V(x)) \\ \text{lev}(V, a) &= \text{lev}(V(a)) \\ \text{lev}(V, ()) &= \text{Low} \\ \text{lev}((\vec{C})^\ell) &= \ell \\ \text{lev}(\text{Lock}(\vec{C})^\ell) &= \ell \\ \text{lev}(\text{Unit}^\ell) &= \ell \end{aligned}$$

Logs are an important part of controlling the complexity of the transaction calculus. In general configurations in the semantics have components of the form $\vec{t} P$, reflecting that every process P executes with respect to a (nested) transaction \vec{t} . We refer to components of this form as *agents* A . The operational semantics are made relatively simple by separating the evolution of process execution from the meta-reasoning about when operational steps are enabled, and what information must be logged to enable the computation.

There are various forms of log rules added during evaluation:

1. A log entry of the form $k::\vec{t} \hat{a} \vec{c}$ requires the sending of a message or generation of a lock. If the former, the message is sent by a transaction operating within the transaction \vec{t} . If the latter, the type system requires that the lock be generated at top-level, outside the scope of any transaction ($|\vec{t}| = 0$).
2. A log entry of the form $k::\vec{t} \check{a} \vec{c}$ records the receipt of a message or acquisition of a lock by a process executing in the transaction \vec{t} .
3. A log entry of the form $k_1 \searrow k_2$ relates a receive event k_2 to the corresponding send event k_1 . It has several purposes. One is to establish a failure dependency from the sending to the receiving

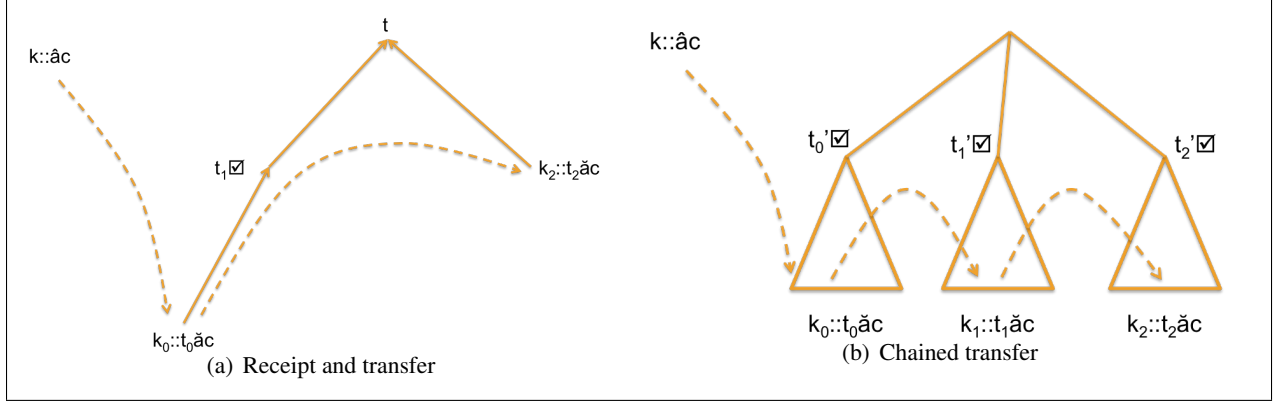


Figure 5: Receipt of message/lock and transfer of ownership

transaction: If the former is aborted, the latter is required in turn to abort. Another purpose is to relate a receive event to the corresponding send event, so that if the latter is undone in the process of aborting a transaction, the corresponding message to be restored is identified.

4. A log entry of the form k undone denotes that the action logged with event identifier k in the logs has been undone. This corresponds to a message receive or lock acquisition event that has been undone because the corresponding transaction has aborted. This type of log entry is used to ensure that message receives are only undone once in the event that a transaction aborts.
5. A log entry of the form $k_2 \curvearrowright k_1$ denotes that the lock acquired in the event labelled with k_2 has been released (or “anti-inherited”) from a transaction to one of its ancestor transactions, as a result of the former transaction having committed. The lock was then acquired by a descendant of that ancestor transaction, in a lock acquisition event labelled k_1 . The actual anti-inheritance of locks up the transaction tree is implicit in the committal of ancestor transactions, and fresh log entries for a lock are only added when descendant of one of these ancestors (a “cousin“ transaction) acquires the lock.. A log entry reflecting the ownership of this lock by the cousin transaction is recorded in the logs with an event identifier k_1 . It is the counterpart to the $k_2 \searrow k_1$, but where the lock has already been acquired in the transaction tree and now its ownership is being transferred within that tree.

Consider the following example (we insert semi-colons for readability):

$$A_0 = (\hat{a} \mid t_1(\check{a}; (\hat{a} \mid \square)) \mid t_2(\check{a}; (\hat{a} \mid \square)) \mid t_3(\check{a}; (\hat{a} \mid \square)))$$

where t_1 , t_2 and t_3 are top-level transactions. Assume that a is an ordinary communication channel. In this case, none of the transactions are nested.

This agent expression may evolve to the following:

$$A_1 = (t_2(\check{a}; (\hat{a} \mid \square)) \mid t_3(\check{a}; (\hat{a} \mid \square)) \mid \llbracket \mathcal{L}_1 \rrbracket)$$

$$\mathcal{L}_1 = k_0::\hat{a} \wedge k_1::t_1\check{a} \wedge k_0 \searrow k_1 \wedge t_1\square$$

and then to:

$$A_2 = (t_3(\check{a}; (\hat{a} \mid \square)) \mid \llbracket \mathcal{L}_2 \rrbracket)$$

$$\mathcal{L}_2 = \mathcal{L}_1 \wedge k'_1::\hat{a} \wedge k_2::t_2\check{a} \wedge k'_1 \searrow k_2 \wedge t_2\square$$

and finally to an empty agent expression $A_3 = \llbracket \mathcal{L}_3 \rrbracket$, with A_3 , with

$$\mathcal{L}_3 = \mathcal{L}_2 \wedge k'_2 :: \hat{a} \wedge k_3 :: t_3 \check{a} \wedge k'_2 \searrow k_3 \wedge t_3 \square.$$

The logs record the receipt of a message by t_1 , the receipt of an unrelated message by t_2 , and the receipt of yet another unrelated message by t_3 . Although these messages are unrelated, they establish failure dependencies between otherwise unrelated transactions. Since a transactions messages not visible until it commits, these failure dependencies are somewhat pointless at this point. A transaction can only establish a failure dependency on a transaction that has already committed.

Consider now the same example, but where a is a lock rather than a message channel. None of the transactions can “send” on the lock channel, since locks can only be sent at top-level, outside the scope of a transaction. Instead, by committing, they implicitly make the lock they have acquired available to other transactions, by anti-inheritance to their parent or to the top-level. So we rewrite the example as follows:

$$A_0 = (\hat{a} \mid t_1(\check{a}; \square) \mid t_2(\check{a}; \square) \mid t_3(\check{a}; \square))$$

This agent expression may evolve to the following:

$$\begin{aligned} A_1 &= (t_2(\check{a}; \square) \mid t_3(\check{a}; \square) \mid \llbracket \mathcal{L}_1 \rrbracket) \\ \mathcal{L}_1 &= k_0 :: \hat{a} \wedge k_1 :: t_1 \check{a} \wedge k_0 \searrow k_1 \wedge t_1 \square \end{aligned}$$

and then to:

$$\begin{aligned} A_2 &= (t_3(\check{a}; \square) \mid \llbracket \mathcal{L}_2 \rrbracket) \\ \mathcal{L}_2 &= \mathcal{L}_1 \wedge k_2 :: t_2 \check{a} \wedge k_1 \curvearrowright k_2 \wedge t_2 \square \end{aligned}$$

and finally to an empty agent expression A_3 , with $A_3 = \llbracket \mathcal{L}_3 \rrbracket$, with

$$\mathcal{L}_3 = \mathcal{L}_2 \wedge k_3 :: t_3 \check{a} \wedge k_2 \curvearrowright k_3 \wedge t_3 \square.$$

The logs record the acquisition of the lock by t_1 , transfer of the lock from t_1 to t_2 , and then to t_3 . This is tracing the history of the transfer of ownership of a single lock through the system.

To understand the lock ownership relations further, consider the example in Fig. 5(a). In this example, a lock $\hat{a} \vec{c}$ generated by the event k has been acquired by a transaction \vec{t}_0 . Once some ancestor transaction \vec{t}_1 has committed (this may be \vec{t}_0 itself), then the lock is available to any descendant of the transaction \vec{t} that is the parent of \vec{t}_1 . If the lock is subsequently acquired by a transaction \vec{t}_2 , then this constitutes a transfer of ownership of the lock from \vec{t}_0 to \vec{t}_2 . In particular we have log entries of the form $k \searrow k_0$ and $k_0 \curvearrowright k_2$. Log entries of the form $k_1 \searrow k_2$ and $k_2 \curvearrowright k_1$ are used to define a relation of “transfer of ownership” of locks, due to a chain of message receive / lock acquisition events. The judgement form for this relation is denoted by $V, \mathcal{L} \vdash k_1 \overset{*}{\curvearrowright} k_2$. The rules for this relation are defined in Fig. 9. So for example we have $V, \mathcal{L} \vdash k \overset{*}{\curvearrowright} k_2$, denoting the transfer of ownership of the lock by the above two acquisition events. In general we may have a chain of lock ownership transfers as transactions acquire locks, then implicitly release them as they commit, as depicted in Fig. 5.

The log entries of the form $k_1 \searrow k_2$ and $k_2 \curvearrowright k_1$ are the basis for an “ownership transfer” relation for locks, $V, \mathcal{L} \vdash k_1 \overset{*}{\curvearrowright} k_2$, given by the rules in Fig. 9. This rules define the transfer of ownership of a lock from one transaction to another, where ownership is transferred via a sequence of lock acquisitions and lock releases (“anti-inheritance”). The log entries of the form $k_1 \searrow k_2$, that denotes a message received or a lock acquired, in combination with the path prefix relation between transaction names, is used to define a failure dependency relationship between transactions. A transaction may be failure dependent on another if the

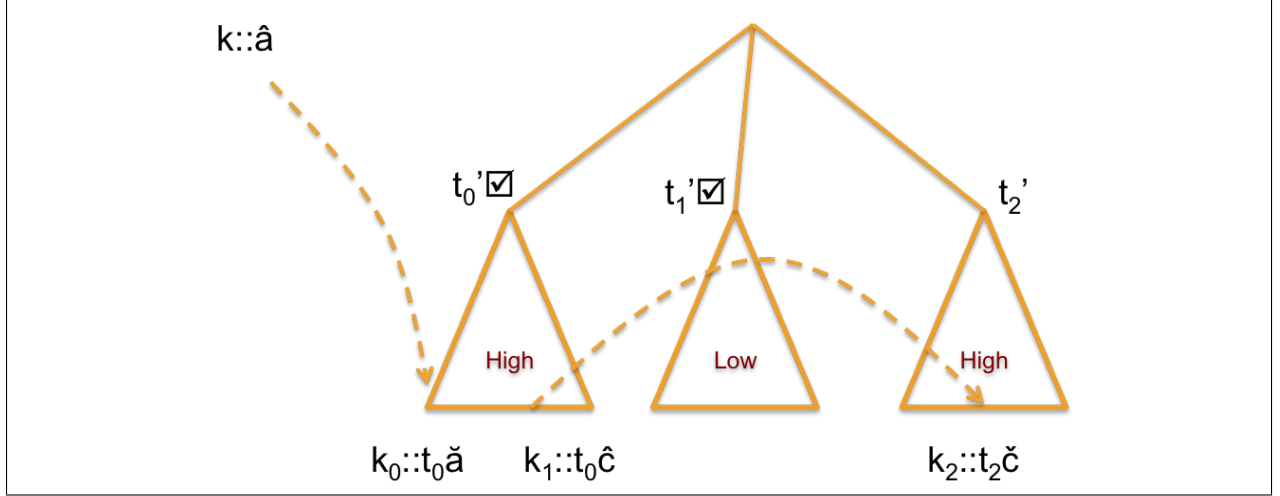


Figure 6: Failure dependencies with retroactive abort

former is a child transaction of the latter, or if the former receives a message from the latter. The rules for failure dependency are provided in Fig. 8.

As already noted, in the absence of retroactive abort, the semantics of nested transactions ensures that the only failure dependencies of interest are those from a parent to a child transaction. The messages sent by a transaction are not visible to other transactions until the transaction commits, and then its messages are made available only to its siblings. The only way the committed transaction can abort is if its parent (or one of its ancestors) aborts, and in that case any siblings that might have become failure dependent on it must also abort. So failure dependencies owing to message exchange are subsumed by the failure dependencies induced by the transaction hierarchy.

For example, consider the following:

$$A_0 = (\hat{a} \mid t_0(t_1[(\check{a}; \hat{b} \mid \square)] \mid t_2[(\check{b}; \square)] \mid \boxtimes))$$

This may evolve to the expression $A_1 = t_0\boxtimes$, with the log

$$\begin{aligned} \mathcal{L}_1 &\equiv k_0::\hat{a} \wedge k_1::t_0.t_1\check{a} \wedge k_0 \searrow k_1 \\ \mathcal{L}_2 &\equiv k_2::t_0.t_1\hat{b} \wedge k_3::t_0.t_2\check{b} \wedge k_2 \searrow k_3 \\ \mathcal{L} &\equiv \mathcal{L}_1 \wedge \mathcal{L}_2 \wedge t_0.t_1\square \wedge t_0.t_2\square. \end{aligned}$$

The transaction $t_0.t_2$ has acquired a failure dependency on its sibling $t_0.t_1$ because of the message on channel b . The transaction $t_0.t_1$ has committed (its messages were not visible before it committed), but it is still possible for one of its ancestors to abort. This happens when t_0 aborts, but since it is the parent of $t_0.t_2$ as well, the latter also aborted. So although $t_0.t_2$ has a failure dependency on its sibling $t_0.t_1$, it will be forced to fail by the abort of the shared parent t_0 , a parent that the siblings must share in order for $t_0.t_2$ to see the output of $t_0.t_1$ before commit of the root transaction.

This situation changes markedly with retroactive abort. Consider the example in Fig. 6. The high transaction \vec{t}_0 acquires the lock generated at event k . The event k_1 corresponds to the sending of a (null) message on channel c . This message became visible to the sibling high transaction \vec{t}_2 once \vec{t}_0 has committed, and event k_2 corresponds to the receipt of this message. The latter induces a failure dependency from \vec{t}_0 to \vec{t}_2 . The intermediate low sibling transaction \vec{t}_1 is unaffected by this. Both it and its low parent are unaware that the latter has “anti-inherited” ownership of the lock a from \vec{t}_0 when it committed.

If another low transaction now seeks to acquire the lock a , retroactive abort will force the abort of \vec{t}_0 , so that ownership of the lock by its parent is removed from the log. This in turn induces the abort of the high sibling \vec{t}_2 , because of the failure dependency induced by message receipt. In all of this, the common parent for \vec{t}_0 and \vec{t}_2 is unaffected, as indeed it should not be since the low transaction \vec{t}_1 should be unaffected. In this way, retroactive abort brings new significance to failure dependencies induced by message exchange.

A crucial point to note here is that acquisition of a lock by a transaction does *not* induce a failure dependency from transaction owning the lock to the transaction acquiring the lock. In the example in Fig. 6, if \vec{t}_1 acquires a lock from \vec{t}_0 , then the abort of \vec{t}_0 does not induce the abort of \vec{t}_1 . This is due to the restricted access to locks provided to transactions: a transaction cannot duplicate or destroy a lock. The type rules require that locks are generated at the top level, outside any transaction. Once acquired, a log entry records the holding of the lock by the transaction, until abort or commit of that transaction makes it available to other transactions. The release of the lock is guaranteed by the semantics of abort and commit of transactions. *In effect we guarantee that locks are handled in a linear fashion: once acquired, a lock is always released. Rather than relying on linear types to statically enforce the linear handling of locks, we rely on the semantics of transactions to enforce this handling.*

The operational semantics of **Tau**_{One} uses various judgements to check preconditions by reference to the log:

$V, \mathcal{L} \vdash k::A$	Identifiable log entry
$V, \mathcal{L} \vdash A$	Anonymous log entry
$V, \mathcal{L} \vdash k_1 \searrow k_2$	Msg or lock acquired
$V, \mathcal{L} \vdash k_1 \curvearrowright k_2$	Lock released
$V, \mathcal{L} \vdash k \text{ undone}$	Action undone
$V, \mathcal{L} \vdash \vec{t} \text{ running}$	Transaction still running
$V, \mathcal{L} \vdash \vec{t} \text{ aborted}$	Transaction aborted
$V, \mathcal{L} \vdash \vec{t} \text{ committed } \vec{t}_0$	Transaction committed
$V, \mathcal{L} \vdash k::A \text{ terminal}$	Terminal lock ownership
$V, \mathcal{L} \vdash k::A \text{ undoable}$	Undoable receive
$V, \mathcal{L} \vdash k::A \text{ transferable } \vec{t}$	Transferable lock
$V, \mathcal{L} \vdash k::A \text{ preemptible } \vec{t}_2 \curvearrowright \vec{t}_1$	Preemptible trans
$V, \mathcal{L} \vdash k_1 \rightsquigarrow k_2$	Failure dependency
$V, \mathcal{L} \vdash k_1 \overset{*}{\curvearrowright} k_2$	Transfer of ownership

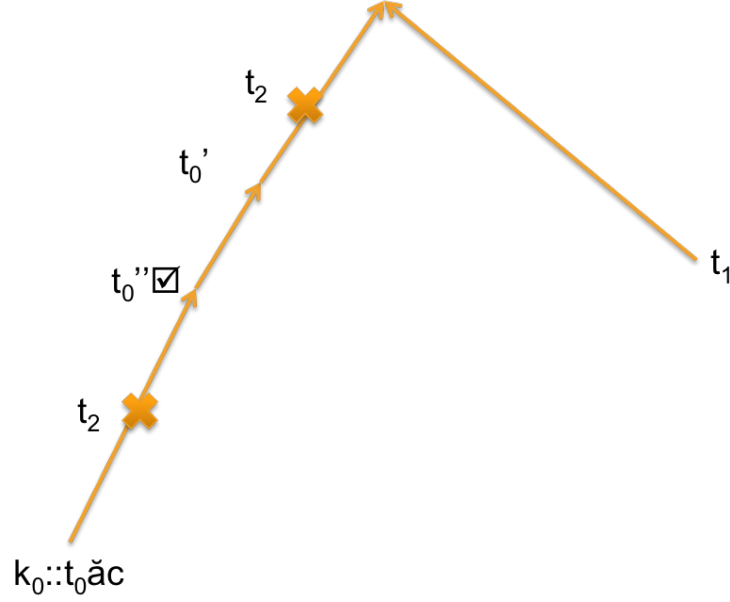
The first five judgements correspond to simply looking up a log entry, while the remaining judgements are based on inferences drawn from the contents of the log. The inference rules for these judgements are provided in Fig. 7:

1. The (LOGJ RUNNING) rule ensures that the corresponding transaction has not committed or aborted. A transaction is committed if it or any of its ancestor transactions has committed, and it has not aborted. A transaction is aborted if it or any transaction upon which it has a failure dependency has aborted. Both of these conditions are checked by the inference rule.
2. The (LOG ABORTED) rule ensures that the corresponding transaction has not aborted. Again this may be due to an action of a process in the transaction itself, or in an ancestor transaction. Recall that the commit or abort of a transaction is independent of that of any of its descendant transactions.
3. The (LOGJ COMMITTED) rule for committed transactions reflects the fact that commit in this model is tentative until all ancestor transactions have committed. The judgement form $V, \mathcal{L} \vdash \vec{t} \text{ committed } \vec{t}_0$

only checks that a transaction \vec{t} has committed up through (a child of) some ancestor \vec{t}_0 . So we have $\vec{t}_0.t' \leq \vec{t}$, i.e., \vec{t}_0 is a prefix of the sequence \vec{t} . The abort of any transaction in the sequence \vec{t}_0 will still cause \vec{t} to abort.

4. The (LOGJ TERMINAL) rule ensures that a lock that has been acquired by a transaction, is still held by that transaction. If the ownership of the lock has been transferred to any other transactions, they must be aborted. In the latter case, ownership of the lock will have transferred back to the original owner.
5. The (LOGJ UNDOABLE) rule is used to determine if a message receive action is undoable. This is the case if the message receive action has been logged, the transaction in which the action was done has aborted (directly or via the actions of an ancestor transaction), and the action is not recorded as having already been undone. If the receive action is undone, an entry reflecting that fact will be added to the log.
6. The (LOGJ TRANSFERABLE) rule is used to determine if a lock that has been acquired, and is being “anti-inherited” as transactions commit, can be released to a transaction seeking to acquire this lock. This is the case if the lock acquisition operation has been logged, the acquired lock has risen sufficiently far in the transaction tree (due to commits) that it is visible to the transaction seeking the lock, and the lock has not already been released to another transaction. If the lock is released, a log entry attesting to this will be added to the log.
7. The (LOGJ PREEMPTIBLE) rule is used to determine if a lock that is held by a high transaction t_0 can be pre-empted by a low transaction t_1 . It must be the case that the lock has not been released to another transaction (i.e., it is terminal), and that it is not already visible to the low transaction seeking the lock ($\vec{t}'_0 \not\leq \vec{t}_1$, where \vec{t}'_0 is the highest this lock has risen in the transaction tree due to commits). This rule identifies the high ancestor transaction t_2 of the lock-holding transaction t_0 that is closest to the root of the overall transaction tree, and selects t_2 for preemptive abort in order to free the lock for t_0 .

To understand the (LOGJ PREEMPTIBLE) rule, consider the following example:



A lock has been acquired by high transaction \vec{t}_0 at event k_0 , and an ancestor transaction \vec{t}_0'' has committed. \vec{t}_0' is the parent of \vec{t}_0'' , so the released lock is available to \vec{t}_0' or any of its child transactions. The requesting low transaction \vec{t}_1 is not a child (or descendant) of \vec{t}_0' ($\vec{t}_0' \not\leq \vec{t}_1$). Retroactive abort is authorized by these circumstances. \vec{t}_2 is the high ancestor transaction of \vec{t}_0 that is closest to the root of the overall transaction, and it is chosen as the transaction to abort. If \vec{t}_2 is between \vec{t}_0' and the root, then \vec{t}_0' and all of its descendant transactions will be aborted. If \vec{t}_2 is between \vec{t}_0 and \vec{t}_0' not including \vec{t}_0' , on the other hand, then \vec{t}_0' must be low and will be unaffected by the abort of \vec{t}_2 and its descendants.

4 Constrained Operational Semantics

In this section, we consider the operational semantics of **TauOne**. An obvious and straightforward way to make use of logs is add preconditions to some of the operational semantics rules that check the logs for certain conditions. For example, a transaction should no longer be able to receive messages from other transactions if it has committed or aborted.

The complication is with our decision to use observational equivalence to reason about noninterference in the presence of concurrent processes with synchronization. A standard property that is required of observational equivalence is that it is preserved under the composition of processes with an arbitrary context. This is the basis for compositional reasoning about process equivalence, which is in turn the basis for reasoning about noninterference. Essentially we show that a “high” process in this semantics is indistinguishable from an empty process that does nothing, under an equality theory that only considers interaction on low channels (including locks).

We are forced to treat logs as local and distributed among the processes, rather than parameterizing the semantics by a single global log, because compositional reasoning about processes includes the ability to encapsulate local channel names within a process, preventing their “leakage” to outside processes unless they are explicitly communicated. *Scope extrusion* in the process equivalence rules is a fundamental aspect of the pi-calculus. Some log entries contain reference to channel names, so they cannot simply be globalized.

$\frac{\mathcal{L} \equiv \mathcal{L}_0 \wedge k:: \vec{t} P}{V, \mathcal{L} \vdash k:: \vec{t} P}$	(LOGJ ENTRY)
$\frac{V, \mathcal{L} \vdash k:: A}{V, \mathcal{L} \vdash A}$	(LOGJ ANON)
$\frac{\forall \vec{t}_0. \vec{t}_0 \leq \vec{t} \supset V, \mathcal{L} \not\vdash \vec{t}_0 \square \quad \forall \vec{t}_0. V, \mathcal{L} \vdash \vec{t}_0 \rightsquigarrow \vec{t} \supset V, \mathcal{L} \not\vdash \vec{t}_0 \boxtimes}{V, \mathcal{L} \vdash \vec{t} \text{ running}}$	(LOGJ RUNNING)
$\frac{\vec{t} = \vec{t}_0.t'_1 \dots t'_m, m > 0 \vec{t}_0.t' \leq \vec{t} \quad V, \mathcal{L} \vdash \vec{t}_0.t' \square \quad \forall \vec{t}_1. V, \mathcal{L} \vdash \vec{t}_1 \rightsquigarrow \vec{t} \supset V, \mathcal{L} \not\vdash \vec{t}_1 \boxtimes \quad V, \mathcal{L} \not\vdash \vec{t}_1 \boxtimes \text{ for any } V, \mathcal{L} \vdash \vec{t}_1 \rightsquigarrow \vec{t} \vec{t}_0 \leq \vec{t}_1 \leq \vec{t}}{V, \mathcal{L} \vdash \vec{t} \text{ committed } \vec{t}_0}$	(LOGJ COMMITTED)
$\frac{V, \mathcal{L} \vdash \vec{t}_0 \boxtimes \quad V, \mathcal{L} \vdash \vec{t}_0 \rightsquigarrow \vec{t} \text{ for some } \vec{t}_0 \leq \vec{t} \quad V, \mathcal{L} \vdash \vec{t}_0 \rightsquigarrow \vec{t}}{V, \mathcal{L} \vdash \vec{t} \text{ aborted}}$	(LOGJ ABORTED)
$\frac{V, \mathcal{L} \vdash k:: \vec{t} \check{a} \vec{c} \quad \forall k_0. k \neq k_0 \wedge V, \mathcal{L} \vdash k \check{r}^* k_0 \supset V, \mathcal{L} \vdash \text{tid}(V, k_0) \text{ aborted}}{V, \mathcal{L} \vdash k:: \vec{t} \check{a} \vec{c} \text{ terminal}}$	(LOGJ TERMINAL)
$\frac{V, \mathcal{L} \vdash k:: \vec{t} \check{a} \vec{c} \text{ terminal} \quad V, \mathcal{L} \vdash \vec{t} \text{ aborted} \quad V, \mathcal{L} \not\vdash k \text{ undone}}{V, \mathcal{L} \vdash k:: \vec{t} \check{a} \vec{c} \text{ undoable}}$	(LOGJ UNDOABLE)
$\frac{V, \mathcal{L} \vdash k:: \vec{t} \check{a} \vec{c} \text{ terminal} \quad V, \mathcal{L} \vdash k_1:: \vec{t} \check{a} \vec{c} \quad V, \mathcal{L} \vdash \vec{t} \text{ committed } \vec{t}_0 \quad \exists k_2. V, \mathcal{L} \vdash k_1 \check{r} k_2}{V, \mathcal{L} \vdash k:: \vec{t} \check{a} \vec{c} \text{ transferable } \vec{t}_0}$	(LOGJ TRANSFERABLE)
$\frac{V, \mathcal{L} \vdash k_0:: \vec{t}_0 \check{a} \vec{c} \text{ terminal} \quad \exists k. V, \mathcal{L} \vdash k_0 \check{r} k \quad V, \mathcal{L} \vdash \vec{t}_0 \text{ committed } \vec{t}'_0 \quad \vec{t}'_0 \not\leq \vec{t}_1 \quad V, \mathcal{L} \vdash \vec{t}'_0 \text{ running} \quad V \vdash \vec{t}_2 \text{ tid}^{\text{Low}} \quad \vec{t}_1 = \min \{ \vec{t} \leq \vec{t}_0 \mid V \vdash \vec{t} \text{ tid}^{\text{High}} \} \text{ lev}(V, \vec{t}_1) = \text{Low} \quad \vec{t}_2 = \min \{ \vec{t} \leq \vec{t}_0 \mid \text{lev}(V, \vec{t}) = \text{High} \}}{V, \mathcal{L} \vdash k_0:: \vec{t}_0 \check{a} \vec{c} \text{ preemptible } \vec{t}_2 \check{r} \vec{t}_1}$	(LOGJ PREEMPTIBLE)

Figure 7: Reasoning From Logs

$V, \mathcal{L} \vdash X \rightsquigarrow X$	(FDEP REFL)
$\frac{V, \mathcal{L} \vdash k_1 \searrow k_2}{V, \mathcal{L} \vdash k_1 \rightsquigarrow k_2}$	(FDEP SENTTO)
$\frac{V, \mathcal{L} \vdash k_1 \curvearrowright k_2}{V, \mathcal{L} \vdash k_1 \rightsquigarrow k_2}$	(FDEP RELEASEDTo)
$\frac{\vec{t}_1 \leq \vec{t}_2}{V, \mathcal{L} \vdash \vec{t}_1 \rightsquigarrow \vec{t}_2}$	(FDEP TID)
$\frac{V(k) = \text{Event}(\vec{t}, \ell) \quad V, \mathcal{L} \vdash k \rightsquigarrow Y}{V, \mathcal{L} \vdash \text{tid}(V, k) \rightsquigarrow Y}$	(FDEP ELEFT)
$\frac{V(k) = \text{Event}(\vec{t}_2, \ell) \quad V, \mathcal{L} \vdash X \rightsquigarrow k}{V, \mathcal{L} \vdash X \rightsquigarrow \text{tid}(V, k)}$	(FDEP ERIGHT)
$\frac{V, \mathcal{L} \vdash X \rightsquigarrow Y \quad V, \mathcal{L} \vdash Y \rightsquigarrow Z}{V, \mathcal{L} \vdash X \rightsquigarrow Z}$	(FDEP TRANS)

Figure 8: Failure Dependency

$V, \mathcal{L} \vdash k \overset{*}{\rightsquigarrow} k$	(XFER REFL)
$\frac{V, \mathcal{L} \vdash k_1 \searrow k_2}{V, \mathcal{L} \vdash k_1 \overset{*}{\rightsquigarrow} k_2}$	(XFER ACQUIRE)
$\frac{V, \mathcal{L} \vdash k_1 \curvearrowright k_2}{V, \mathcal{L} \vdash k_1 \overset{*}{\rightsquigarrow} k_2}$	(XFER TRANSFER)
$\frac{V, \mathcal{L} \vdash k_1 \overset{*}{\rightsquigarrow} k_2 \quad V, \mathcal{L} \vdash k_2 \overset{*}{\rightsquigarrow} k_3}{V, \mathcal{L} \vdash k_1 \overset{*}{\rightsquigarrow} k_3}$	(XFER TRANS)

Figure 9: Ownership Dependency

$\frac{V(a) = (\vec{C})^\ell \quad tid(V, k_1) = \vec{t} \quad V, \mathcal{L} \vdash k_1 \searrow k_2 \quad V, \mathcal{L} \vdash k_2 :: \vec{t} \check{a} \vec{c}}{V, \mathcal{L} \Vdash k_1 :: \vec{t} \hat{a} \vec{c}}$	(WFLOG MSG)
$\frac{V(a) = \text{Lock}(\vec{C})^\ell \quad tid(V, k_1) = \vec{t} \quad V, \mathcal{L} \vdash k_1 \searrow k_2 \quad V, \mathcal{L} \vdash k_2 :: \vec{t} \check{a} \vec{c} \quad V, \mathcal{L} \vdash k :: \vec{t} \check{a} \vec{c} \text{ terminal} \quad V, \mathcal{L} \vdash k_1 \overset{*}{\rightsquigarrow} k \quad \forall k'. \left(\begin{array}{l} \mathcal{A} \vdash k_1 \overset{*}{\rightsquigarrow} k' \not\vdash k_1 \overset{*}{\rightsquigarrow} k' \text{ or } \mathcal{A} \vdash k' :: \vec{t} \check{a} \vec{c} \text{ terminal} \\ \not\vdash k' :: \vec{t} \check{a} \vec{c} \text{ terminal} \\ \text{or } V, \mathcal{L} \vdash tid(V, k') \text{ aborted or } k' = k \end{array} \right)}{V, \mathcal{L} \Vdash k_1 :: \vec{t} \hat{a} \vec{c}}$	(WFLOG LOCK HELD)
$\frac{V(a) = \text{Lock}(\vec{C})^\ell \quad tid(V, k_1) = \vec{t} \quad V, \mathcal{L} \vdash k_1 \searrow k_2 \quad V, \mathcal{L} \vdash k_2 :: \vec{t} \check{a} \vec{c} \quad \forall k'. \left(\begin{array}{l} \mathcal{A} \vdash k_1 \overset{*}{\rightsquigarrow} k' \not\vdash k_1 \overset{*}{\rightsquigarrow} k' \text{ or } \mathcal{A} \vdash k' :: \vec{t} \check{a} \vec{c} \text{ terminal} \\ \not\vdash k' :: \vec{t} \check{a} \vec{c} \text{ terminal} \\ \text{or } V, \mathcal{L} \vdash tid(V, k') \text{ aborted} \end{array} \right)}{V, \mathcal{L} \Vdash k_1 :: \vec{t} \hat{a} \vec{c}}$	(WFLOG LOCK FREE)
$\frac{tid(V, k_2) = \vec{t} \quad V, \mathcal{L} \vdash k_1 \searrow k_2 \quad V, \mathcal{L} \vdash k_1 :: \vec{t} \hat{a} \vec{c}}{V, \mathcal{L} \Vdash k_2 :: \vec{t} \check{a} \vec{c}}$	(WFLOG RECV)
$\frac{tid(V, k_2) = \vec{t} \quad V, \mathcal{L} \vdash k_1 \rightsquigarrow k_2 \quad V, \mathcal{L} \vdash k_1 :: \vec{t} \check{a} \vec{c} \quad V(a) = \text{Lock}(\vec{C})^\ell}{V, \mathcal{L} \Vdash k_2 :: \vec{t} \check{a} \vec{c}}$	(WFLOG ACQ)
$\frac{V, \mathcal{L} \vdash k_1 :: \vec{t} \hat{a} \vec{c} \quad V, \mathcal{L} \vdash k_2 :: \vec{t} \check{a} \vec{c}}{V, \mathcal{L} \Vdash k_1 \searrow k_2}$	(WFLOG SENTTO)
$\frac{V, \mathcal{L} \vdash k_1 :: \vec{t} \check{a} \vec{c} \quad V, \mathcal{L} \vdash k_2 :: \vec{t} \check{a} \vec{c} \quad V(a) = \text{Lock}(\vec{C})^\ell}{V, \mathcal{L} \Vdash k_1 \rightsquigarrow k_2}$	(WFLOG XFERTO)
$\frac{V, \mathcal{L} \vdash k_1 :: \vec{t}_1 \hat{a} \vec{c} \quad V, \mathcal{L} \vdash k_1 \searrow k_2 \quad V, \mathcal{L} \vdash k_2 :: \vec{t}_2 \check{a} \vec{c} \quad \vec{t} \boxtimes}{V, \mathcal{L} \Vdash k_2 \text{ undone}}$	(WFLOG UNDONE)
$V, \mathcal{L} \Vdash \vec{t} \boxtimes$	(WFLOG COMMIT)
$V, \mathcal{L} \Vdash \vec{t} \boxtimes$	(WFLOG ABORT)

Figure 10: **Tau**_{Zero} and **Tau**_{One}: Well Formed Logs

Since transaction state, as represented by logs, is now local, compositional reasoning in turns requires some mechanism for constraining the possible contexts. For example, if we allow a transaction to make progress because we assume it is still running, we must constrain the surrounding context to prevent any log entries that record it as committed or aborted, or record any causal dependencies on transactions that are committed or aborted.

An LTS is a standard tool in reasoning about process equivalence. In a traditional LTS, transition rules are given labels that specify an observable behavior exhibited by a process as part of a reaction rule. For example, the label may specify that the process has offered the behavior of sending or receiving a message.

Our purpose is different, and therefore we define the LTS differently. The purpose of an LTS here is to give definitions of the reduction rules for the semantics that are purely local. The labels in the transition rules then specify logical conditions that must be satisfied by any context with which a process performing such an action is composed. For example, if a reduction rule requires as a precondition that a transaction is still running, then any context surrounding the application of that reduction rule must satisfy the conditions:

1. There are no log entries for abort of any transactions upon which that transaction is failure dependent.
2. No ancestor of that transaction can have committed.

We define a logic in which these conditions are specified, and statements in this logic are used as labels in the LTS to constrain contexts surrounding the application of the reduction rules of the semantics:

$$\begin{aligned}
\mathcal{F}^A & ::= \vec{t}_1 \rightsquigarrow \vec{t}_2 \mid k_1 \overset{*}{\rightsquigarrow} k_2 \mid k::\vec{t} \hat{a} \vec{c} \mid k::\vec{t} \check{a} \vec{c} \mid \\
& \quad \vec{t}[\square] \mid k \text{ undone} \mid k_1 = k_2 \\
\mathcal{F} & ::= \text{true} \mid \mathcal{F}_1 \wedge \mathcal{F}_2 \mid \mathcal{F}_1 \vee \mathcal{F}_2 \mid \\
& \quad \forall X:T.\mathcal{F} \mid \exists X:T.\mathcal{F} \mid \mathcal{F}^A \mid \neg \mathcal{F}^A
\end{aligned}$$

We furthermore distinguish the positive and negative constraints:

$$\begin{aligned}
\mathcal{F}^+ & ::= \text{true} \mid \mathcal{F}_1^+ \wedge \mathcal{F}_2^+ \mid \exists X:T.\mathcal{F}^+ \mid \mathcal{F}^A \\
\mathcal{F}^- & ::= \text{true} \mid \mathcal{F}_1^- \wedge \mathcal{F}_2^- \mid \forall X:T.\mathcal{F} \mid \neg \mathcal{F}^A
\end{aligned}$$

Define:

$$\mathcal{F}_1^A \supset \mathcal{F}_2 \equiv \neg \mathcal{F}_1^A \vee \mathcal{F}_2$$

Positive constraints require the presence of particular log entries. Once satisfied, a positive constraint may be discharged. Negative constraints on the other hand prevent certain conditions from becoming true in the logs. They typically contain both universal quantifiers (quantifying over all log entries) and negative conditions (sometimes constraining the domain of a universal quantifier, and sometimes requiring that certain forms of log entries be absent). Since they are intended to constrain all expansions of the logs, it is not possible to discharge them. The key observation is that constraints concerning messages ($k::\vec{t} \hat{a} \vec{c}$ and $k::\vec{t} \check{a} \vec{c}$) are always positive constraints, and thus can eventually be discharged. With transaction identifiers and event identifiers always globally defined, this allows local scoping of message channel and lock names. The latter in turn is important for reasoning about noninterference.

The use of negative conditions is restricted to checking the absence of log entries of particular forms (including failure and ownership dependency). As can be seen from the premises of the inference rules in Fig. 7, this is sufficient for encoding the negative conditions that are required for some of the reduction rules

$$\begin{aligned}
\mathcal{F}(\vec{t} \text{ running}) &= (\forall \vec{t}_0. \vec{t}_0 \leq \vec{t} \supset \neg \vec{t}_0 \boxtimes) \wedge (\forall \vec{t}_0. \vec{t}_0 \rightsquigarrow \vec{t} \supset \neg \vec{t}_0 \boxtimes) \\
\mathcal{F}(\vec{t} \text{ committed } \vec{t}_0) &= \exists t'. (\vec{t}_0.t' \leq \vec{t} \wedge \vec{t}_0.t' \boxtimes) \wedge \forall \vec{t}_1. (\vec{t}_1 \rightsquigarrow \vec{t} \supset \neg \vec{t}_1 \boxtimes) \\
\mathcal{F}(\vec{t} \text{ aborted}) &= \exists \vec{t}_0. \vec{t}_0 \boxtimes \wedge \vec{t}_0 \rightsquigarrow \vec{t} \\
\mathcal{F}(k::\vec{t} \check{a} \vec{c} \text{ terminal}) &= k::\vec{t} \check{a} \vec{c} \wedge \forall k_0. (k = k_0 \vee \neg(k \overset{*}{\rightsquigarrow} k_0) \vee \exists \vec{t}_0. \vec{t}_0 \boxtimes \wedge \vec{t}_0 \rightsquigarrow k_0) \\
\mathcal{F}(k::\vec{t} \hat{a} \vec{c} \text{ undoable}) &= \mathcal{F}(k::\vec{t} \check{a} \vec{c} \text{ terminal}) \wedge \mathcal{F}(\vec{t} \text{ aborted}) \wedge \neg(k \text{ undone}) \\
\mathcal{F}(k::\vec{t} \check{a} \vec{c} \text{ transferable } \vec{t}_0) &= \mathcal{F}(k::\vec{t} \check{a} \vec{c} \text{ terminal}) \wedge \mathcal{F}(\vec{t} \text{ committed } \vec{t}_0) \\
\mathcal{F}(k_0::\vec{t}_0 \check{a} \vec{c} \text{ preemptible } \vec{t}_2 \rightsquigarrow \vec{t}_1) &= \mathcal{F}(k_0::\vec{t}_0 \check{a} \vec{c} \text{ terminal}) \wedge \\
&\quad \exists \vec{t}'_0. (\mathcal{F}(\vec{t}'_0 \text{ committed } \vec{t}'_0) \wedge \mathcal{F}(\vec{t}'_0 \text{ running}))
\end{aligned}$$

Figure 11: Mapping Log Judgements to Constraints

in the semantics. We define the mapping from judgements of the log inference logic to logical preconditions in this constrained reduction semantics, in Fig. 11.

Given an agent expression A , then A^* denotes the logs underlying this agent expression, i.e., all log entries that are contained in log expressions in the agent expression.

$$\begin{aligned}
(\vec{t} P)^* &= \text{true} \\
\llbracket \mathcal{L} \rrbracket^* &= \mathcal{L} \\
(A_1 | A_2)^* &= A_1^* \wedge A_2^* \\
((\nu a:C)A)^* &= \exists a:C.A^*
\end{aligned}$$

The existential quantifier is used to abstract over locally scoped channel names in the logs. We also define A^\dagger , the underlying agent expression with logs removed:

$$\begin{aligned}
(\vec{t} P)^\dagger &= \vec{t} P \\
\llbracket \mathcal{L} \rrbracket^\dagger &= \text{stop} \\
(A_1 | A_2)^\dagger &= A_1^\dagger | A_2^\dagger \\
((\nu a:C)A)^\dagger &= (\nu a:C)A^\dagger
\end{aligned}$$

The constrained semantics are specified in Fig. 13 using reduction rules of the form:

$$V \vdash A \xrightarrow{\mathcal{F}} A'$$

and in Fig. 14 using reaction rules of the form:

$$\begin{aligned}
(V, A) &\xrightarrow{\mathcal{F}} (V', A') \\
(V, A) &\xrightarrow{\mathcal{F}} (V', A')
\end{aligned}$$

The former rules denote ‘‘internal’’ reduction steps within a process expression, while the latter reaction rules denote computational steps that involve interactions with the surrounding context. In both forms of rules, the set V records type bindings of global channels, the agent expression A denotes the redex, and the agent expression A' denotes the reduct. The reaction rules include a constraint on the surrounding logs in the context,

as already explained. We use $\left\{ \begin{array}{l} V \vdash A \Longrightarrow A' \\ (V, A) \longrightarrow (V', A') \end{array} \right\}$ as an abbreviation for $\left\{ \begin{array}{l} V \vdash A \xrightarrow{\text{true}} A' \\ (V, A) \xrightarrow{\text{true}} (V', A') \end{array} \right\}$

$$\begin{array}{c}
A \equiv A \mid \text{stop} \\
\text{stop} \equiv (va:C)\text{stop} \\
\vec{t} \text{ stop} \equiv \text{stop} \\
A_1 \mid A_2 \equiv A_2 \mid A_1 \\
A_1 \mid (A_2 \mid A_3) \equiv (A_1 \mid A_2) \mid A_3 \\
((va:C)A_1) \mid A_2 \equiv (va:C)(A_1 \mid A_2), \quad a \notin \text{fn}(A_2) \\
(va_1:C_1)(va_2:C_2)A \equiv (va_2:C_2)(va_1:C_1)A \\
[[\mathcal{L}_1 \wedge \mathcal{L}_2]] \equiv [[\mathcal{L}_1]] \mid [[\mathcal{L}_2]] \\
A \equiv A \\
\frac{A_1 \equiv A_2}{A_2 \equiv A_1} \\
\frac{A_1 \equiv A_2 \quad A_2 \equiv A_3}{A_1 \equiv A_3} \\
\mathcal{L} \equiv \mathcal{L} \wedge \text{true} \\
\mathcal{L}_1 \wedge \mathcal{L}_2 \equiv \mathcal{L}_2 \wedge \mathcal{L}_1 \\
\mathcal{L}_1 \wedge (\mathcal{L}_2 \wedge \mathcal{L}_3) \equiv (\mathcal{L}_1 \wedge \mathcal{L}_2) \wedge \mathcal{L}_3 \\
\mathcal{L} \equiv \mathcal{L} \vee \text{false} \\
\mathcal{L}_1 \vee \mathcal{L}_2 \equiv \mathcal{L}_2 \vee \mathcal{L}_1 \\
\mathcal{L}_1 \vee (\mathcal{L}_2 \vee \mathcal{L}_3) \equiv (\mathcal{L}_1 \vee \mathcal{L}_2) \vee \mathcal{L}_3 \\
\mathcal{L} \equiv \mathcal{L} \\
\frac{\mathcal{L}_1 \equiv \mathcal{L}_2}{\mathcal{L}_2 \equiv \mathcal{L}_1} \\
\frac{\mathcal{L}_1 \equiv \mathcal{L}_2 \quad \mathcal{L}_2 \equiv \mathcal{L}_3}{\mathcal{L}_1 \equiv \mathcal{L}_3}
\end{array}$$

Figure 12: Structural Equivalence Rules

$\frac{A_1 \equiv A_2}{V \vdash A_1 \Longrightarrow A_2}$	(RED REFL)
$\frac{V \vdash A_1 \xrightarrow{\mathcal{F}_1^-} A_2 \quad V \vdash A_2 \xrightarrow{\mathcal{F}_2^-} A_3}{V \vdash A_1 \xrightarrow{\mathcal{F}_1^- \wedge \mathcal{F}_2^-} A_3}$	(RED TRANS)
$V \vdash \vec{t}((v=v) \rightarrow P_1 \parallel P_2) \Longrightarrow \vec{t} P_1$	(RED IFTRUE)
$\frac{v_1 \neq v_2}{V \vdash \vec{t}((v_1=v_2) \rightarrow P_1 \parallel P_2) \Longrightarrow \vec{t} P_2}$	(RED IFFALSE)
$V \vdash \vec{t}(\text{repl } P) \Longrightarrow \vec{t}(P \mid \text{repl } P)$	(RED REPL)
$V \vdash \vec{t}(P_1 \mid P_2) \Longrightarrow (\vec{t} P_1 \mid \vec{t} P_2)$	(RED FORK)
$V \vdash \vec{t}(\vec{t}_0[P]) \Longrightarrow (\vec{t}, \vec{t}_0)P$	(RED TRANS)
$\frac{\mathcal{F} = \mathcal{F}(k_1::\vec{t}_1 \hat{a} \vec{c} \text{ undoable}) \wedge k_2 \searrow k_1 \wedge k_2::\vec{t}_2 \hat{a} \vec{c}}{V \vdash \text{stop} \xrightarrow{\mathcal{F}} (\vec{t}_2 \hat{a} \vec{c} \mid \llbracket k_1 \text{ undone} \rrbracket)}$	(RED UNDO)
$\frac{(V_1.(a:C)) \vdash A'_1 \xrightarrow{\mathcal{F}} A'_2 \quad a \notin \text{fn}(\mathcal{F})}{V_1 \vdash (va:C)A_1 \xrightarrow{\mathcal{F}} (va:C)A_2}$	(RED NEW)
$\frac{V_1 \vdash A_1 \xrightarrow{\mathcal{F}} A'_1 \quad V, A_2 \vdash \mathcal{F}^-}{V_1 \vdash (A_1 \mid A_2) \xrightarrow{\mathcal{F}} (A'_1 \mid A_2)}$	(RED PAR)
$\frac{V_1 \vdash A_1 \xrightarrow{\mathcal{F}_1} A_2 \quad \mathcal{F} \equiv \mathcal{F}_1 \wedge \mathcal{F}_2}{V_1 \vdash A_1 \xrightarrow{\mathcal{F}} A_2}$	(RED SUB)

Figure 13: Constrained Reduction Rules

$(V, \vec{t} \square) \xrightarrow{\mathcal{F}(\vec{t} \text{ running})} (V, \llbracket \vec{t} \square \rrbracket) V \vdash \vec{t} \square \xrightarrow{\mathcal{F}(\vec{t} \text{ running})} \llbracket \vec{t} \square \rrbracket$	(REACT FINISH)
$(V, \vec{t} (\text{await } t_0[\square] \text{ then } P)) \xrightarrow{(\vec{t}.t_0)[\square]} (V, \vec{t} P)$	(REACT AWAIT)
$ \begin{array}{c} A_1 = \vec{t}_1 (\check{a} \vec{x} P_1 + P_2) \quad A_2 = \vec{t}_2 \hat{a} \vec{c} \\ \vec{t}_0 \leq \vec{t}_1 \text{ and } \vec{t}_0 \leq \vec{t}_2 \text{ for some } \vec{t}_0 \quad \mathcal{F}_1 = \mathcal{F}(\vec{t}_1 \text{ running}) \quad \mathcal{F}_2 = \mathcal{F}(\vec{t}_2 \text{ committed } \vec{t}_0) \\ V' = V.(k_1 : \text{Event}(\vec{t}_1, \text{lev}(V, \vec{t}_1))).(k_2 : \text{Event}(\vec{t}_2, \text{lev}(V, \vec{t}_2))) \\ \mathcal{L}' = k_1 :: \vec{t}_1 \check{a} \vec{c} \wedge k_2 :: \vec{t}_2 \hat{a} \vec{c} \wedge k_2 \setminus k_1 \end{array} $ <hr style="width: 100%;"/> $(V, (A_1 A_2)) \xrightarrow{\mathcal{F}_1 \wedge \mathcal{F}_2} (V', (\vec{t}_1 \{ \vec{c} / \vec{x} \} P_1 \llbracket \mathcal{L}' \rrbracket))$	(REACT SYNC)
$ \begin{array}{c} A \equiv \vec{t}_1 (\check{a} \vec{x} P_1 + P_2) \quad \mathcal{F}_1 = \mathcal{F}(\vec{t}_1 \text{ running}) \quad V(a) = \text{Lock}(\vec{C})^\ell \text{ for some } \vec{C}, \ell \\ \vec{t}_0 \leq \vec{t}_1 \text{ and } \vec{t}_0 \leq \vec{t}_2 \text{ for some } \vec{t}_0 \quad \mathcal{F}_2 = \mathcal{F}(k_2 :: \vec{t}_2 \check{a} \vec{c} \text{ transferable } \vec{t}_0) \\ V' = V.(k_1 : \text{Event}(\vec{t}_1, \text{lev}(V, \vec{t}_1))) \quad \mathcal{L}' = k_2 \curvearrowright k_1 \wedge k_1 :: \vec{t} \check{a} \vec{c} \end{array} $ <hr style="width: 100%;"/> $(V, A) \xrightarrow{\mathcal{F}_1 \wedge \mathcal{F}_2} (V', (\vec{t}_1 \{ \vec{c} / \vec{x} \} P_1 \llbracket \mathcal{L}' \rrbracket))$	(REACT TRANSFER)
$ \begin{array}{c} A \equiv \vec{t}_1 (\check{a} \vec{x} P_1 + P_2) \quad V(a) = \text{Lock}(\vec{C})^\ell \text{ for some } \vec{C}, \ell \\ \mathcal{F}_1 = \mathcal{F}(\vec{t}_1 \text{ running}) \quad \mathcal{F}_2 = \mathcal{F}(k_0 :: \vec{t}_0 \check{a} \vec{c} \text{ preemptible } \vec{t}_2 \curvearrowright \vec{t}_1) \end{array} $ <hr style="width: 100%;"/> $(V, A) \xrightarrow{\mathcal{F}_1 \wedge \mathcal{F}_2} (V, (A \llbracket \vec{t}_2 \boxtimes \rrbracket))$	(REACT PREEMPT)
$ \begin{array}{c} ((V_1.(a : C), A'_1) \xrightarrow{\mathcal{F}} (V_2.(a : C), A'_2) \quad a \notin \text{fn}(\mathcal{F})) \\ (V_1, (va : C) A_1) \xrightarrow{\mathcal{F}} (V_2, (va : C) A_2) \end{array} $	(REACT NEW)
$ \begin{array}{c} (V_1, A_1) \xrightarrow{\mathcal{F}} (V_2, A'_1) \quad V, A_2^* \vdash \mathcal{F}^- \\ (V_1, (A_1 A_2)) \xrightarrow{\mathcal{F}} (V_2, (A'_1 A_2)) \end{array} $	(REACT PAR)
$ \begin{array}{c} V_1 \vdash A_1 \xrightarrow{\mathcal{F}_1^-} A'_1 \quad (V_1, A'_1) \xrightarrow{\mathcal{F}^-} (V_2, A'_2) \quad V_1 \vdash A'_2 \xrightarrow{\mathcal{F}_2^-} A_2 \\ (V_1, A_1) \xrightarrow{\mathcal{F}_1^- \wedge \mathcal{F}^- \wedge \mathcal{F}_2^-} (V_2, A_2) \end{array} $	(REACT STRUCT)
$ \begin{array}{c} (V_1, A_1) \xrightarrow{\mathcal{F}_1} (V_2, A_2) \quad \mathcal{F} \equiv \mathcal{F}_1 \wedge \mathcal{F}_2 \\ (V_1, A_1) \xrightarrow{\mathcal{F}} (V_2, A_2) \end{array} $	(REACT SUB)

Figure 14: Constrained Reaction Rules

$\frac{V_1 \vdash A_1 \xrightarrow{\exists X:T.\mathcal{F}} A_2 \quad V(Y) = T \quad V, A_1^* \vdash \{Y/X\}.\mathcal{F}}{V_1 \vdash A_1 \xrightarrow{\{Y/X\}.\mathcal{F}} A_2}$	(RED DISCHARGE EXISTS)
$\frac{V_1 \vdash A_1 \xrightarrow{\mathcal{F}_1^+ \wedge \mathcal{F}_2} A_2 \quad V, A_1^* \vdash \mathcal{F}_1^+}{V_1 \vdash A_1 \xrightarrow{\mathcal{F}_2} A_2}$	(RED DISCHARGE AND)
$\frac{(V_1, A_1) \xrightarrow{\exists X:T.\mathcal{F}} (V_2, A_2) \quad V(Y) = T \quad V, A_1^* \vdash \{Y/X\}.\mathcal{F}}{(V_1, A_1) \xrightarrow{\{Y/X\}.\mathcal{F}} (V_2, A_2)}$	(REACT DISCHARGE EXISTS)
$\frac{(V_1, A_1) \xrightarrow{\mathcal{F}_1^+ \wedge \mathcal{F}_2} (V_2, A_2) \quad V, A_1^* \vdash \mathcal{F}_1^+}{(V_1, A_1) \xrightarrow{\mathcal{F}_2} (V_2, A_2)}$	(REACT DISCHARGE AND)

Figure 15: Discharge Rules

The logs are implicit in the collection of agents, and the condition \mathcal{F} places global conditions on the logs that must be satisfied by any context. For example, the (REACT SYNC) rule for synchronization adds the log entries recording the sending and receiving of a message, but adds the constraints that the receiving process is still running and that the sending process' messages have been sufficiently committed to be visible to the receiving process.

There is no syntactic distinction between messages and locks. A lock that is available to be acquired is represented by an atom of the form $\vec{r} \hat{a} \vec{c}$, while a process that is attempting to acquire the lock a (with parameters $\{\vec{x}\}$) is represented by a process of the form $\vec{r} \check{a} \vec{x} P$. The (RED SYNC) rule for receiving a message also suffices for acquiring a lock. When a lock is acquired, the logs are extended in such a way that if the acquiring transaction subsequently aborts, the acquired lock can be released back to the original generator of the lock instance (via the (RED UNDO) computation rule).

We provide more details on the semantic distinction between messages and locks in Sect. 6, when we consider a security type system. In Fig. 14, a difference between messages and locks can be seen in the (REACT TRANSFER) rule for transferring locks held by committed transactions. This rule requires that the logs contain a record of a lock acquisition. The rule explicitly checks, via the type of the bound name a , that it is a lock rather than a message channel. The (RED UNDO) reduction rule, for releasing a lock held by an aborted transaction, releases a received message or acquired lock back to the original sender. The (REACT TRANSFER) rule on the other hand transfers a lock to some ‘‘cousin’’ transaction that has requested the lock, by releasing the lock to some common ancestor that has committed, thus enabling the cousin transaction to acquire the lock. The fact that the common ancestor has committed ensures that the changes made by the releasing transaction can be made visible to the acquiring transaction.

The discharge rules are part of the operational semantics, and are provided in Fig. 15. These rules allow the discharging of positive constraints that are implied by the visible log entries. This is appropriate since the positive constraints test for the presence of particular log entries. The negative constraints on the other hand prevent certain log entries in the surrounding context. The consistency of the logs with the negative constraints is tested outside of the operational semantics.

The definition of a context \mathcal{C} is given by:

$$\mathcal{C} ::= [] \mid (\mathcal{C} \mid A) \mid (\nu a:C)\mathcal{C}$$

$V, \mathcal{F} \vdash \mathcal{F}$	(CON HYPOTH)
$\frac{V, \mathcal{F}_i \vdash \mathcal{F}}{V, \mathcal{F}_1 \wedge \mathcal{F}_2 \vdash \mathcal{F}}$	(CON ANDL)
$\frac{V, \mathcal{F} \vdash \mathcal{F}_1 \quad V, \mathcal{F} \vdash \mathcal{F}_2}{V, \mathcal{F} \vdash \mathcal{F}_1 \wedge \mathcal{F}_2}$	(CON ANDR)
$\frac{V, \mathcal{F} \vdash \mathcal{F}_i}{V, \mathcal{F} \vdash \mathcal{F}_1 \vee \mathcal{F}_2}$	(CON ORR)
$\frac{V(Y) = T \quad V, \mathcal{F} \vdash \{Y/X\}\mathcal{F}}{V, \mathcal{F} \vdash \exists X:T.\mathcal{F}}$	(CON EXISTS)
$\frac{\bigwedge \{V, \mathcal{F} \vdash \{Y/X\}\mathcal{F} \mid V(Y) = T\}}{V, \mathcal{F} \vdash \forall X:T.\mathcal{F}}$	(CON FORALL)
$\frac{\vec{t}_1 \not\leq \vec{t}_2}{V, \mathcal{F} \vdash \neg(\vec{t}_1 \leq \vec{t}_2)}$	(CON NLEQ)
$V, \mathcal{F} \vdash X \neq X$	(CON EQ)
$\frac{V, \mathcal{F} \not\vdash \vec{t}_1 \rightsquigarrow \vec{t}_2}{V, \mathcal{F} \vdash \neg(\vec{t}_1 \rightsquigarrow \vec{t}_2)}$	(CON NFDEP)
$\frac{V, \mathcal{F} \not\vdash k_1 \overset{*}{\rightsquigarrow} k_2}{V, \mathcal{F} \vdash \neg(k_1 \overset{*}{\rightsquigarrow} k_2)}$	(CON NODEP)
$\frac{V, \mathcal{F} \not\vdash \vec{t}[\square]}{V, \mathcal{F} \vdash \neg \vec{t}[\square]}$	(CON NSTAT)
$\frac{V, \mathcal{F} \not\vdash k \text{ undone}}{V, \mathcal{F} \vdash \neg(k \text{ undone})}$	(CON NUNDONE)

Figure 16: **Tau_{one}**: Constraint Reasoning

An expression of the form $\mathcal{C}[A]$ denotes instantiating a context with an agent.

The rules for reasoning about constraints are provided in Fig. 16. The universal quantifier rules are extensional, quantifying over all transaction and event identifiers. In practice the preconditions only need to be checked for identifiers present in the logs, so the premises are in fact finitary. The negative conditions use a form of “negation as failure” to test if a condition is not derivable from the logs. We omit the rules for failure and ownership dependency, which are the same as in Fig. 8 and Fig. 9. We also omit the usual weakening and contraction rules.

Since the semantics includes evaluation under a context, we also extend reasoning to reasoning about constraints under a context:

Definition 4.1 *Given a context \mathcal{C} , define the rules for the judgement form $V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{C}[\mathcal{F}]$ by:*

$$\frac{V, \mathcal{L} \wedge A_2^* \vdash \mathcal{C}[A_1] \triangleright \mathcal{C}[\mathcal{F}]}{V, \mathcal{L} \vdash (\mathcal{C}[A_1] \mid A_2) \triangleright (\mathcal{C}[\mathcal{F}] \mid A_2)} \quad (\text{CTXT PAR})$$

$$\frac{(V.(a : C), \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{C}[\mathcal{F}])}{V, \mathcal{L} \vdash \mathcal{C}[(va:C)A] \triangleright \mathcal{C}[(va:C)\mathcal{F}]} \quad (\text{CTXT NEW})$$

$$\frac{V, \mathcal{L} \wedge A^* \vdash \mathcal{F}}{V, \mathcal{L} \vdash A \triangleright \mathcal{F}} \quad (\text{CTXT AGENT})$$

Lemma 4.1 *Assume $\mathcal{C} \equiv (v \vec{a} : \vec{C})([] \mid A_0)$. Then*

$$V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{C}[\mathcal{F}] \text{ iff } (V. \vec{a} : \vec{C}), \mathcal{L} \wedge A^* \wedge A_0^* \vdash \mathcal{F}.$$

PROOF: A straightforward induction on the height of the derivation \mathcal{C} . □

We also have a notion of *contextual constraint entailment*. We use the judgement form

$$V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F}_1 \supset \mathcal{C}[\mathcal{F}_2].$$

The idea is that a computation step may have occurred with the agent A , in the context \mathcal{C} . The original set of constraints required to justify this computation step was \mathcal{F}_2 . The constraint set \mathcal{F}_1 represents the result of simplifying these constraints under the context \mathcal{C} . For example, a positive constraint in \mathcal{F}_2 requiring the presence of a particular log entry may be discharged if that log entry is present in A . If the remaining constraints \mathcal{F}_1 are satisfied by the agent $\mathcal{C}[A]$, augmented by the log entries \mathcal{L} , then the original set of constraints \mathcal{F}_2 is satisfiable by the logs in A , the log entries added by the context \mathcal{C} , and the log entries \mathcal{L} . The inference rules for the logic of contextual constraint entailment are provided in Fig. 17.

The following lemma states the following: Suppose that \mathcal{F}_1 is the constraint set derived from a computation step in a context \mathcal{C} , modified by the discharge rules. Let \mathcal{L}_0 be any extension of the logs that is compatible with this constraint set, when combined with the existing logs. Note that \mathcal{L}_0 may contain log entries that are necessary in order to discharge some of the positive constraints. Fig 17.

Lemma 4.2 *Suppose $V. \vec{a} : \vec{C} \vdash A_1 \xrightarrow{\mathcal{F}_2} A_2$ and $V \vdash \mathcal{C}[A_1] \xrightarrow{\mathcal{F}_1} \mathcal{C}[A_2]$. If there is some $V_0 \supseteq V$ and \mathcal{L}_0 such that $V_0, \mathcal{L}_0 \vdash \mathcal{F}_1$ then there is a derivation for $V_0, \mathcal{L}_0 \vdash \mathcal{C}[A_1] \triangleright \mathcal{F}_1 \supset \mathcal{C}[\mathcal{F}_2]$.*

$\frac{V, A^* \vdash \mathcal{F}^+}{V, \mathcal{L} \vdash A \triangleright \text{true} \supset \mathcal{F}^+}$	(CE DIS)
$\frac{\mathcal{F} = \mathcal{F}^+ \wedge \mathcal{F}^- \quad V, \mathcal{L} \wedge A^* \vdash \mathcal{F}^+}{V, \mathcal{L} \vdash A \triangleright \mathcal{F}^- \supset \mathcal{F}}$	(CE CON)
$\frac{V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F}'_1 \supset \mathcal{C}[\mathcal{F}'_2] \quad \mathcal{F}_1 \equiv \mathcal{F}'_1 \quad \mathcal{F}_2 \equiv \mathcal{F}'_2}{V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F}_1 \supset \mathcal{C}[\mathcal{F}_2]}$	(CE EQUIV)
$\frac{V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F}_1 \supset \mathcal{C}[\{Y/X\}.\mathcal{F}_2] \quad V(Y) = T}{V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F}_1 \supset \mathcal{C}[\exists X:T.\mathcal{F}_2]}$	(CE EXISTSR)
$\frac{V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}_1] \quad V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}_2]}{V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}_1 \wedge \mathcal{F}_2]}$	(CE ANDR)
$\frac{V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F}_i \supset \mathcal{C}[\mathcal{F}]}{V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F}_1 \wedge \mathcal{F}_2 \supset \mathcal{C}[\mathcal{F}]}$	(CE ANDL)
$\frac{V, \mathcal{L} \wedge A_0^* \vdash \mathcal{C}[A] \triangleright \mathcal{F}_1 \supset \mathcal{C}[\mathcal{F}_2]}{V, \mathcal{L} \vdash \mathcal{C}[A \mid A_0] \triangleright \mathcal{F}_1 \supset \mathcal{C}[\mathcal{F}_2]}$	(CE PAR)
$\frac{V.a : C, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F}_1 \supset \mathcal{C}[\mathcal{F}_2]}{V, \mathcal{L} \vdash \mathcal{C}[(va:C)A] \triangleright \mathcal{F}_1 \supset \mathcal{C}[\mathcal{F}_2]}$	(CE NEW)

Figure 17: Rules for Contextual Constraint Entailment

PROOF: We verify the result by induction on the height of the context \mathcal{C} . We use the derivation for $V_0, \mathcal{L}_0 \vdash \mathcal{F}_1$ in the base cases, after using the (CE EXISTSR) and (CE ANDR) to analyze the positive constraints in the original constraint for the computation step. The (CE DIS) rule corresponds to the analysis of a positive constraint that was discharged (so there must have been a derivation for the positive constraint in the derivation for $V \vdash \mathcal{C}[A_1] \xrightarrow{\mathcal{F}_1} \mathcal{C}[A_2]$). \square

The following lemma relates contextual constraint entailment to contextual satisfiability for constraints: Suppose the constraint set \mathcal{F}_1 results from propagating the original constraint set \mathcal{F}_2 through the context, and discharging some of the constraints. Suppose this former constraint set is satisfied by the logs seen so far, augmented with the log extension \mathcal{L} . Then the original constraints \mathcal{F}_2 , generated by the computation step in the context \mathcal{C} , are also satisfied with this log extension.

Lemma 4.3 *Suppose $V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F}_1 \supset \mathcal{C}[\mathcal{F}_2]$. If $V, \mathcal{L} \wedge \mathcal{C}[A] \vdash \mathcal{F}_1$, then $V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{C}[\mathcal{F}_2]$.*

PROOF: By induction on the derivation of $V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F}_1 \supset \mathcal{C}[\mathcal{F}_2]$. \square

Lemma 4.4 *Suppose $V, \mathcal{L} \vdash \mathcal{C}[A] \triangleright \mathcal{F}_1 \supset \mathcal{C}[\mathcal{F}_2]$. If $V, \mathcal{L} \vdash \mathcal{C}_0[\mathcal{C}[A]] \triangleright \mathcal{C}_0[\mathcal{F}_1]$, then $V, \mathcal{L} \vdash \mathcal{C}_0[\mathcal{C}[A]] \triangleright \mathcal{C}_0[\mathcal{C}[\mathcal{F}_2]]$.*

PROOF: By induction on the height of the context \mathcal{C}_0 , using the previous lemma in the base case. \square

$$\begin{array}{c}
\frac{}{\text{Low} \preceq \text{High}} \quad \frac{}{T \preceq T} \quad \frac{T_1 \preceq T_2 \quad T_2 \preceq T_3}{T_1 \preceq T_3} \\
\hline
\frac{\ell_1 \preceq \ell_2}{\text{Lock}(\vec{C})^{\ell_1} \preceq \text{Lock}(\vec{C})^{\ell_2}}
\end{array}$$

Figure 18: Type System: Inclusions

Finally we have the following notion of repeated computation:

$$\begin{aligned}
(V, A) \xrightarrow{* \mathcal{F}^-} (V', A') \quad \text{iff} \quad & V = V_0, A = A_0, V' = V_n, A' = A_n, \\
& (V_j, A_j) \xrightarrow{\mathcal{F}_j^-} (V_{j+1}, A_{j+1}) \text{ for } j = 0, \dots, n-1 \\
& \text{and } \mathcal{F}^- \equiv \mathcal{F}_0^- \wedge \dots \wedge \mathcal{F}_{n-1}^-,
\end{aligned}$$

Once a reduction step has taken place, a negative condition may no longer be true in the resulting configuration. For example, once the receipt of a message by an aborted transaction has been undone, the absence of a log entry recording this undoing is no longer true. However the constraints on the reduction steps are intended to constrain the observer rather than the computation (once the positive constraints have been discharged). An allowable observer context is one in which all of the undischarged negative constraints are satisfied. Any changes to the logs that invalidates one of these constraints is internal to the process being observed.

5 Security Type System

In this section, we provide a security type system for nested transactions. A type system is specified for this language using several judgement forms:

$\ell_1 \preceq \ell_2$	Inclusion of security levels
$T_1 \preceq T_2$	Type inclusion
$V \vdash \text{env}$	Well formed context
$V \vdash \mathcal{L} \text{ log}$	Well formed log
$V \vdash T \text{ type}$	Well formed type
$V \vdash \vec{T} \text{ tid}^\ell$	Well formed transaction id
$V \vdash v : T$	Well formed value
$V \vdash A \text{ agent}^\ell$	Well formed agent
$V \vdash \vec{T} P \text{ proc}^\ell$	Well formed process

The type system is provided in Fig. 18, Fig. 19, Fig. 20, Fig. 21 and Fig. 22. Mainly of note here is that subtyping on channel names is invariant on the security level of the channel. Effectively this partitions high and low processes into separate subnets so that they cannot communicate with each other. We have explained earlier why this is necessary.

Lemma 5.1 *If $V \vdash A_1 \text{ agent}^\ell$ and $A_1 \equiv A_2$, then $V \vdash A_2 \text{ agent}^\ell$.*

PROOF: By induction on derivation of Agents (rules in Fig. 20), along with the case analysis of structural equivalence rules (Fig. 12). For example, to prove the case (AGENT NEW), in terms of structural equivalence,

$\varepsilon \vdash \text{env}$	(ENV NULL)
$\frac{V \vdash \text{env} \quad V \vdash T \text{ type}}{V.(v : T) \vdash \text{env}}$	(ENV EXTEND)
$\frac{\overrightarrow{V \vdash C \text{ type}} \quad V \vdash \text{env}}{V \vdash (\overrightarrow{C})^\ell \text{ type}}$	(TYCON CHAN)
$\frac{\overrightarrow{V \vdash C \text{ type}} \quad V \vdash \text{env}}{V \vdash \text{Lock}(\overrightarrow{C})^\ell \text{ type}}$	(TYCON LOCK)
$\frac{V \vdash \text{env}}{V \vdash \text{Trans}(\ell) \text{ type}}$	(TYCON TRANS)
$\frac{V \vdash \text{env} \quad V \vdash \vec{t} \text{ tid}^\ell}{V \vdash \text{Event}(\vec{t}, \ell) \text{ type}}$	(TYCON EVENT)
$\frac{V \vdash \text{env}}{V \vdash \text{Event}(\varepsilon, \ell) \text{ type}}$	(TYCON EVENT)
$\frac{\overrightarrow{V \vdash t : \text{Trans}(\ell)} \quad \ell_i \preceq \ell_j \text{ for } 1 \leq i \leq j \leq n = \vec{t} \quad V \vdash \text{env}}{V \vdash \vec{t} \text{ tid}^{\ell_n}}$	(TRANS ID)

Figure 19: Type System: Type Constructors

$\frac{V \vdash A_1 \text{ agent}^\ell \quad V \vdash A_2 \text{ agent}^\ell}{V \vdash (A_1 A_2) \text{ agent}^\ell}$	(AGENT PAR)
$\frac{V.(a : C) \vdash A \text{ agent}^\ell}{V \vdash (\nu a : C) A \text{ agent}^\ell}$	(AGENT NEW)
$\frac{V \vdash \vec{t} \text{ tid}^\ell \quad V \vdash \vec{t} P \text{ proc}^\ell}{V \vdash \vec{t} P \text{ agent}^\ell}$	(AGENT PROC)
$\frac{V \vdash \mathcal{L} \text{ log}}{V \vdash \llbracket \mathcal{L} \rrbracket \text{ agent}^\ell}$	(AGENT LOG)
$\frac{V \vdash A \text{ agent}^{\ell_0} \quad \ell \preceq \ell_0}{V \vdash A \text{ agent}^\ell}$	(AGENT SUB)

Figure 20: Type System: Agents

$\frac{V \vdash \text{env}}{V \vdash a : V(a)}$	(VAL NAME)
$\frac{V \vdash \text{env}}{V \vdash x : V(x)}$	(VAL VAR)
$\frac{V \vdash v : T \quad T \preceq T'}{V \vdash v : T'}$	(VAL SUB)

Figure 21: Type System: Values

$\frac{V \vdash v : (\vec{C})^\ell \quad \overline{V \vdash v : \vec{C}}}{V \vdash \vec{t} \hat{v} \vec{v} \text{ proc}^\ell}$	(PROC SEND)
$\frac{V \vdash v : \text{Lock}(\vec{C})^\ell \quad \overline{V \vdash v : \vec{C}}}{V \vdash \hat{v} \vec{v} \text{ proc}^\ell}$	(PROC LOCK)
$\frac{V \vdash a : (\vec{C})^\ell \quad \overline{V.(\vec{x} : \vec{C}) \vdash \vec{t} P \text{ proc}^\ell}}{V \vdash \vec{t} \sum \{\vec{a} \vec{x} P\} \text{ proc}^\ell}$	(PROC RECV)
$\frac{ \vec{t} > 0 \quad \overline{V \vdash a : \text{Lock}(\vec{C})^\ell} \quad \overline{V.(\vec{x} : \vec{C}) \vdash \vec{t} P \text{ proc}^\ell}}{V \vdash \vec{t} \sum \{\vec{a} \vec{x} P\} \text{ proc}^\ell}$	(PROC ACQUIRE)
$\frac{ \vec{t} > 0 \quad V \vdash \vec{t} \text{ tid}^\ell}{V \vdash \vec{t} \square \text{ proc}^\ell}$	(PROC FINISH)
$\frac{V \vdash \vec{t}.t_0 \text{ tid}^{\ell_0} \quad \ell_0 \preceq \ell \quad V \vdash \vec{t} P \text{ proc}^\ell}{V \vdash \vec{t} (\text{await } t_0 [\square] \text{ then } P) \text{ proc}^\ell}$	(PROC AWAIT)
$\frac{V(v_1) = V(v_2) \quad \text{lev}(V, v_1) = \ell \quad V \vdash \vec{t} P_1 \text{ proc}^\ell \quad V \vdash \vec{t} P_2 \text{ proc}^\ell}{V \vdash \vec{t} ((v_1 = v_2) \rightarrow P_1 [] P_2) \text{ proc}^\ell}$	(PROC CHOICE)
$\frac{V \vdash \vec{t} P \text{ proc}^\ell}{V \vdash \vec{t} (\text{repl } P) \text{ proc}^\ell}$	(PROC REPL)
$\frac{V \vdash \vec{t} P_1 \text{ proc}^\ell \quad V \vdash \vec{t} P_2 \text{ proc}^\ell}{V \vdash \vec{t} (P_1 P_2) \text{ proc}^\ell}$	(PROC FORK)
$\frac{V \vdash \vec{t}.t_0 \text{ tid}^{\ell_0} \quad \ell \preceq \ell_0 \quad V \vdash (\vec{t}.t_0) P \text{ proc}^{\ell_0}}{V \vdash \vec{t} (\vec{t}_0 [P]) \text{ proc}^\ell}$	(PROC TRANS)
$\frac{V \vdash \vec{t} P \text{ proc}^{\ell_0} \quad \ell \preceq \ell_0}{V \vdash \vec{t} P \text{ proc}^\ell}$	(PROC SUB)

Figure 22: Type System: Processes

without lose of generality we suppose A_1 is $(\nu a:C)(A | A')$, then A_2 is $((\nu a:C)A) | A'$, where $a \notin \text{fn}(A')$. By induction, we derive $V.(a:C) \vdash (A | A') \text{ agent}^\ell$, again by rule (AGENT PAR), we get $V.(a:C) \vdash A \text{ agent}^\ell$ and $V.(a:C) \vdash A' \text{ agent}^\ell$. The former concludes $V \vdash (\nu a:C)A \text{ agent}^\ell$ based on rule (AGENT NEW), similarly the latter concludes $V \vdash A' \text{ agent}^\ell$ due to $a \notin \text{fn}(A')$. Apply rule (AGENT PAR) again, we derive $V \vdash ((\nu a:C)A | A') \text{ agent}^\ell$, which concludes A_2 . \square

Lemma 5.2 (Type Preservation Under Reduction) *If $V_1 \vdash A_1 \text{ agent}^\ell$ and $V_1 \vdash A_1 \xrightarrow{\mathcal{F}} A_2$ then $V_2 \vdash A_2 \text{ agent}^\ell$.*

PROOF: By induction on reduction rules (Fig. 13), we examine some cases. For example, to prove the case for (RED TRANS): By induction, we derive that $V \vdash A_1 \xrightarrow{\mathcal{F}_1^-} A_2$, $V \vdash A_2 \text{ agent}^\ell$, also $V \vdash A_2 \xrightarrow{\mathcal{F}_2^-} A_3$, $V \vdash A_3 \text{ agent}^\ell$. By an application of (RED SUB), we get $V \vdash A_1 \xrightarrow{\mathcal{F}_1^- \wedge \mathcal{F}_2^-} A_2$ and $V \vdash A_2 \xrightarrow{\mathcal{F}_1^- \wedge \mathcal{F}_2^-} A_3$. By transitivity, we conclude $V \vdash A_1 \xrightarrow{\mathcal{F}_1^- \wedge \mathcal{F}_2^-} A_3$ where $V \vdash A_3 \text{ agent}^\ell$. \square

Lemma 5.3 (Type Preservation Under Substitution) *If $V.\vec{x} : \vec{T} \vdash \vec{t} P \text{ proc}^\ell$ and $V \vdash \vec{a} : \vec{x}$ then $V \vdash \vec{t} \{ \vec{a} / \vec{x} \} P \text{ proc}^\ell$.*

PROOF: By induction on a derivation of the well-typedness of a process (Fig. 22). The proof proceeds based on cases, most of which are similar. We only show the case (PROC RECV) as a demonstration. In order to make syntactical term more distinguishable in this proof case, we slightly change the variable names in the lemma so that it becomes: If $V.\vec{y} : \vec{T} \vdash \vec{t} P \text{ proc}^\ell$ and $V \vdash \vec{b} : \vec{T}$ then $V \vdash \vec{t} \{ \vec{b} / \vec{y} \} P \text{ proc}^\ell$. Then, for the case of (PROC RECV), we have, $V.\vec{y} : \vec{T} \vdash \vec{t} \sum \{ \vec{a} \vec{x} P \} \text{ proc}^\ell$. By induction hypothesis and permutation, we get $V.(\vec{x} : \vec{C}).(\vec{y} : \vec{T}) \vdash \vec{t} P \text{ proc}^\ell$. By weakening, we have $V.\vec{x} : \vec{C} \vdash \vec{b} : \vec{T}$. By induction, $V.\vec{x} : \vec{C} \vdash \vec{t} \{ \vec{b} / \vec{y} \} P \text{ proc}^\ell$. We conclude the result by an application of (PROC RECV). \square

Theorem 5.1 (Type Preservation Under Reaction) *If $V_1 \vdash A_1 \text{ agent}^\ell$ and $(V_1, A_1) \xrightarrow{\mathcal{F}} (V_2, A_2)$ then $V_2 \vdash A_2 \text{ agent}^\ell$.*

PROOF: By induction on the reaction rules (Fig. 14). We present the primary steps for proving the case of the (REACT SYNC) rule as an example: By the assumption that $V \vdash A_1 | A_2 \text{ agent}^\ell$, we derive that $V \vdash \vec{t}_1 (\vec{a} \vec{x} P_1 + P_2) \text{ agent}^\ell$ and $V \vdash \vec{t}_2 \vec{a} \vec{c} \text{ agent}^\ell$. From the premises for (PROC RECV) and (PROC SEND), we have $V.\vec{x} : \vec{C} \vdash \vec{t}_1 P_1 \text{ proc}^\ell$ and $V \vdash \vec{c} : \vec{C}$. Then, by Lemma 5.3 we get $V \vdash \vec{t}_1 \{ \vec{c} / \vec{x} \} P_1 \text{ agent}^\ell$. Similarly, we can derive $V' \vdash \llbracket \mathcal{L}' \rrbracket \text{ agent}^\ell$ because V' records all events type in $\llbracket \mathcal{L}' \rrbracket$. Since $V \subseteq V'$, then by type rule (AGENT PAR), we can conclude that $V' \vdash (\vec{t}_1 \{ \vec{c} / \vec{x} \} P_1 | \llbracket \mathcal{L}' \rrbracket) \text{ agent}^\ell$. \square

6 Noninterference

In this section, we verify noninterference for the calculus of nested transactions, based on the security type system presented in the previous section. The verification is based on the observable behavior of processes.

Because our calculus is asynchronous, we only consider observables arising from messages sent, since the receiving of messages cannot be observed directly.

Messages offered to the environment are defined in terms of barbs. Our definition of barbs differs from the normal case in certain important regards:

1. The definition of barbs is relativized to the transactional level at which an observation may be made. For example, a message cannot be observed at a certain level \vec{t} until that message has become visible due to being committed by lower level transactions.
2. Barbs include a constraint on the context for the output to be offered. For example, a process may only offer a message to the context if any transaction of which it is a part is still running.
3. Barbs may be offered not just by outputting messages, but also by relinquishing locks. Such locks are not manipulated explicitly by the processes, but are recorded in the logs and manipulated implicitly by commit and abort operations, and lock acquisition by other processes.
4. Barbs may be implicitly available, if they correspond to locks that are held by high-level processes that may be preempted by low-level processes.

Definition 6.1 (Strong Barbs) *Define that the agent A in the environment V offers the strong barb $\vec{t} a$ with constraint \mathcal{F}^- , written $(V, A) \downarrow_{\vec{t} a}^{\mathcal{F}^-}$, if $V \vdash A \xrightarrow{\mathcal{F}_1^-} A'$ for some A' and \mathcal{F}_1^- , and there is a context \mathcal{C} such that one of the following cases holds:*

1. $A' = \mathcal{C}[\vec{t}_0 \hat{a} \vec{c}]$ and $\mathcal{F}_2 = \mathcal{F}(\vec{t}_0 \text{ committed } \vec{t})$; or for some \vec{t}_0, \vec{c} ; or
2. $A' = \mathcal{C}[\vec{t}_0 \llbracket \mathcal{L} \rrbracket]$ and $\mathcal{L} \equiv \mathcal{L}' \wedge k :: \vec{t}_0 \check{a} \vec{c}$ and $\mathcal{F}_2 = \mathcal{F}(k :: \vec{t}_0 \check{a} \vec{c} \text{ transferable } \vec{t})$ for some \vec{t}_0, \vec{c} ; or
3. $A' = \mathcal{C}[\vec{t} (\check{a} \vec{x} P_1 + P_2)]$ and $\mathcal{F}_2 = \mathcal{F}(k_0 :: \vec{t}_1 \check{a} \vec{c} \text{ preemptible } \vec{t}_2 \curvearrowright \vec{t}) \wedge \mathcal{F}(\vec{t} \text{ running})$, and $\text{lev}(V, \vec{t}) = \text{Low}$ and $\text{lev}(V, \vec{t}_2) = \text{High}$.

Furthermore we must have

$$V, \text{true} \vdash A' \triangleright \mathcal{F}_2^- \supset \mathcal{F}_2$$

for some \mathcal{F}_2^- such that $\mathcal{F} \equiv \mathcal{F}_1^- \wedge \mathcal{F}_2^-$.

Strong barbs characterize the channels on which an agent expression has committed to offering outputs. For example, although reduction may evaluate an equality test operation, both values being compared are ground by the time this test is considered in the reduction relation, and therefore it is already predetermined which branch will be taken in the test. The reduction rules amount to “book-keeping” to recognize such realities about the computation. None of the structural equivalence or reduction rules introduce or eliminate strong barbs:

Lemma 6.1 *Suppose $V \vdash A_1 \xrightarrow{\mathcal{F}^-} A_2$. Then $(V, A_2) \downarrow_{\vec{t} a}^{\mathcal{F}_2^-}$ if and only if $(V, A_1) \downarrow_{\vec{t} a}^{\mathcal{F}_1^-}$, for some $\mathcal{F}_1^- \equiv \mathcal{F}^- \wedge \mathcal{F}_2^-$.*

PROOF: The “only if” part follows from the definition of strong barbs. For the “if” part, we construct a derivation for $(V, A_2) \downarrow_{\vec{t} a}^{\mathcal{F}_2^-}$ by induction on the derivation for $V \vdash A_1 \xrightarrow{\mathcal{F}^-} A_2$, using the fact that $(V, A_1) \downarrow_{\vec{t} a}^{\mathcal{F}_1^-}$ for the base where the reduction sequence from A_1 to A_2 has zero length. For the inductive step, we verify by case analysis that each reduction step preserves strong barbs. \square

Definition 6.2 (Barbs) Define $(V, A) \Downarrow_{\tau a}^{\mathcal{F}^-}$ if $(V, A) \xrightarrow{* \mathcal{F}_1^-} (V', A')$ and $(V', A') \Downarrow_{\tau a}^{\mathcal{F}_2^-}$, and $\mathcal{F}^- \equiv \mathcal{F}_1^- \wedge \mathcal{F}_2^-$.

Definition 6.3 (Barbed Bisimulation) Define that a relation R is a barbed bisimulation if R is symmetric and, whenever $(A_1, A_2) \in R$, we have:

1. If for some V and \mathcal{F}_1^- we have $(V, A_1) \xrightarrow{\mathcal{F}_1^-} (V', A'_1)$, then $(V, A_2) \xrightarrow{* \mathcal{F}_2^-} (V_2', A'_2)$ for some $V_2', A'_2, \mathcal{F}_2^-$ such that $\mathcal{F}_1^- \equiv \mathcal{F}_2^-$ and $(A'_1, A'_2) \in R$.
2. If for some V and \mathcal{F}_1^- we have $(V, A_1) \Downarrow_{\tau a}^{\mathcal{F}_1^-}$, then $(V, A_2) \Downarrow_{\tau a}^{\mathcal{F}_2^-}$, for some \mathcal{F}_2^- such that $\mathcal{F}_1^- \equiv \mathcal{F}_2^-$.

Define that A_1 and A_2 are bisimilar, written $A_1 \overset{\bullet}{\approx} A_2$, if $(A_1, A_2) \in R$ for some bisimulation R .

A context \mathcal{C} is a (V_1, ℓ_1) - (V_2, ℓ_2) -context if $V_2 \vdash \mathcal{C} \text{ agent}^{\ell_2}$ is derivable from $V_1 \vdash [] \text{ agent}^{\ell_1}$. An environment V is *closed* if it only binds names, and event and transaction identifiers. In particular it does not bind program variables.

Definition 6.4 (Barbed Congruence) Define the barbed congruence $\approx_{V, \ell}$ by: $A_1 \approx_{V, \ell} A_2$ if

1. $V \vdash A_1 \text{ agent}^{\ell}$,
2. $V \vdash A_2 \text{ agent}^{\ell}$, and
3. for any closed V' and secrecy level ℓ' , for any (V, ℓ) - (V', ℓ') -context \mathcal{C} , we have $\mathcal{C}[A_1] \overset{\bullet}{\approx} \mathcal{C}[A_2]$.

Our main tool for reasoning about noninterference is an erasure of processes of high security level. We then show that any barbed congruences among low processes are unaffected by the erasure of the high processes. The complication with this is that there may be interactions between high and low processes. For example, a low process may acquire a lock that was released by a high process after it committed. The key insight is that erasure must be defined in such a way that it modifies the logs to remove any references to logs of high processes. We consider some further explanation of this key point.

Recall the following example:

$$A_0 = (\hat{a} \mid t_1(\check{a}; \square) \mid t_2(\check{a}; \square) \mid t_3(\check{a}; \square))$$

where we assume now that t_1 and t_3 are low transactions, and t_2 is high. Assume that a is a low-level lock, of type $\text{Lock}()^{\text{Low}}$. As we have seen, this agent expression may evolve to the following:

$$\begin{aligned} A_1 &= (t_2(\check{a}; \square) \mid t_3(\check{a}; \square) \mid \llbracket \mathcal{L}_1 \rrbracket) \\ \mathcal{L}_1 &= k_0::\hat{a} \wedge k_1::t_1\check{a} \wedge k_0 \searrow k_1 \wedge t_1 \square \end{aligned}$$

and then to:

$$\begin{aligned} A_2 &= (t_3(\check{a}; \square) \mid \llbracket \mathcal{L}_2 \rrbracket) \\ \mathcal{L}_2 &= \mathcal{L}_1 \wedge k_2::t_2\check{a} \wedge k_1 \curvearrowright k_2 \wedge t_2 \square \end{aligned}$$

and finally to $A_3 = \llbracket \mathcal{L}_3 \rrbracket$, with

$$\mathcal{L}_3 = \mathcal{L}_2 \wedge k_3::t_3\check{a} \wedge k_2 \curvearrowright k_3 \wedge t_3 \square.$$

The logs record the acquisition of the lock by t_1 , transfer of the lock from t_1 to t_2 , and then to t_3 . In the erasure of the logs, we remove the dependency of the low transaction t_3 on the high transaction t_1 by rewriting the log entries to a dependency on t_1 instead:

$$\mathcal{E}_{V, \mathcal{L}}^L(\mathcal{L}_3) = \mathcal{E}_{V, \mathcal{L}}^L(\mathcal{L}_1) \wedge k_3::t_3\check{a} \wedge k_1 \curvearrowright k_3 \wedge t_3 \square.$$

In the erasure, the logs for any high transactions have been removed, and the logs of any low transactions have been modified where necessary to remove any dependencies on the erased high transactions.

Consider now the above example where it is the transaction t_1 , that initially acquires the lock, that is high. Both t_2 and t_3 are low. In the final configuration, we have the logs:

$$\begin{aligned}\mathcal{E}_{V,\varphi}^L(\mathcal{L}_2) &= k_0::\hat{a} \wedge k_2::t_2\check{a} \wedge k_0 \searrow k_2 \wedge t_2 \square. \\ \mathcal{E}_{V,\varphi}^L(\mathcal{L}_3) &= \mathcal{E}_{V,\varphi}^L(\mathcal{L}_2) \wedge k_3::t_3\check{a} \wedge k_2 \curvearrowright k_3 \wedge t_3 \square.\end{aligned}$$

In this case, the logs for t_1 have been erased. These logs contain entries $k_1::t_1\check{a}$ and $k_0 \searrow k_1$. In the erasure, the log entry $k_1 \curvearrowright k_2$ for t_2 is translated to $k_0 \searrow k_2$. The transfer of the a lock from t_1 to t_2 is translated in the log entries into the acquisition of the lock at top level by t_2 .

In general the most complication in the erasure of logs is caused by high transactions acquiring locks from top level, and then releasing them to low transactions. Consider the following modification of the above example:

$$A_0 = (\hat{a} \mid t_1(\check{a}; \square) \mid t_2(\check{a}; \boxtimes) \mid t_3(\check{a}; \square))$$

where t_1 is high, and t_2 and t_3 are low. Again, this agent expression may evolve to the following:

$$\begin{aligned}A_1 &= (t_2(\check{a}; \square) \mid t_3(\check{a}; \square) \mid \llbracket \mathcal{L}_1 \rrbracket) \\ \mathcal{L}_1 &= k_0::\hat{a} \wedge k_1::t_1\check{a} \wedge k_0 \searrow k_1 \wedge t_1 \square\end{aligned}$$

and then to:

$$\begin{aligned}A_2 &= (t_3(\check{a}; \square) \mid \llbracket \mathcal{L}_2 \rrbracket) \\ \mathcal{L}_2 &= \mathcal{L}_1 \wedge k_2::t_2\check{a} \wedge k_1 \curvearrowright k_2 \wedge t_2 \boxtimes\end{aligned}$$

and finally to $A_3 = \llbracket \mathcal{L}_3 \rrbracket$, with

$$\mathcal{L}_3 = \mathcal{L}_2 \wedge k_3::t_3\check{a} \wedge k_1 \curvearrowright k_3 \wedge t_3 \square.$$

In this example, t_3 obtained the lock from t_1 , after t_2 aborted. Therefore the erasure will rewrite the log entry $k_1 \curvearrowright k_3$ to $k_0 \searrow k_3$.

The following result shows that erasure can be permuted with the instantiation of a context. This is a critical tool in the verification of noninterference: We may reason independently about bisimilarity of a process and its erasure, and any context that is composed with that process.

Lemma 6.2 *Suppose that $V \vdash A$ agent ^{ℓ} and \mathcal{C} is a (V, ℓ) - (V', ℓ') context. Then*

$$\mathcal{E}_{V', \mathcal{L}, \mathcal{M}}^A(\mathcal{C}[A]) = \mathcal{E}_{V', \mathcal{L}, \mathcal{M}}^A(\mathcal{C})[\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A)].$$

PROOF: By induction on the structure of \mathcal{C} . □

Lemma 6.3 *Suppose $V \vdash A$ agent ^{ℓ} . If $\text{lev}(V, v) = \text{High}$, then $v \notin \text{fn}(\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A)) \cup \text{fv}(\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A))$.*

We now prove some results about the simulation of an agent A by its erasure $\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A)$.

Lemma 6.4 (Erased Reductions) *If $V \vdash A_1 \xrightarrow{\mathcal{F}} A_2$, then $V \vdash \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_1) \xrightarrow{\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^{\mathcal{F}}} \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_2)$.*

PROOF: A straightforward induction on the derivation of $V \vdash A_1 \xrightarrow{\mathcal{F}} A_2$. □

$$\begin{aligned}
\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_1 \mid A_2) &= (\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_1) \mid \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_2)) \\
\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A((\mathbf{va}:C)A) &= (\mathbf{va}:C)\mathcal{E}_{(V.(a:C)), \mathcal{L}, \mathcal{M}}^A(A) \\
\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(\vec{t}P) &= \begin{cases} \text{stop} & \text{if } lev(V, \vec{t}) = \text{High} \\ \vec{t} \mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(P) & \text{if } lev(V, \vec{t}) = \text{Low} \\ \mathcal{E}_{V, \mathcal{L}, \varepsilon}^P(P) & \text{if } |\vec{t}| = 0 \end{cases} \\
\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A([\mathcal{L}_0]) &= ([\mathcal{E}_{V, \mathcal{L}}^L(\mathcal{L}_0) \wedge \mathcal{U}_{V, \mathcal{L}, \mathcal{M}}(\mathcal{L}_0)] \mid \mathcal{R}_{V, \mathcal{L}, \mathcal{M}}(\mathcal{L}_0)) \\
\mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(\hat{v}\vec{v}) &= \begin{cases} \hat{v}\vec{v} & \text{if } lev(V, v) = \text{Low} \\ \text{stop} & \text{if } lev(V, v) = \text{High} \end{cases} \\
\mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(\check{a}\vec{x}P) &= \begin{cases} \check{a}\vec{x} \mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(P) & \text{if } lev(V, a) = \text{Low} \\ \text{stop} & \text{if } lev(V, a) = \text{High} \end{cases} \\
\mathcal{E}_{V, \mathcal{L}, \vec{t}}^P((v_1=v_2) \rightarrow P_1 \square P_2) &= \begin{cases} (v_1=v_2) \rightarrow \mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(P_1) \square \mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(P_2) & \text{if } lev(V, v_1) = \text{Low} \\ \text{stop} & \text{if } lev(V, v_1) = \text{High} \end{cases} \\
\mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(P_1 + P_2) &= \mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(P_1) + \mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(P_2) \\
\mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(P_1 \mid P_2) &= \mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(P_1) \mid \mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(P_2) \\
\mathcal{E}_{V, \mathcal{L}, \vec{t}}^P((\mathbf{va}:C)P) &= (\mathbf{va}:C)\mathcal{E}_{(V.(a:C)), \mathcal{L}, \vec{t}}^P(P) \\
\mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(\text{repl } P) &= \text{repl } \mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(P) \\
\mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(\text{await } t[\square] \text{ then } P) &= \begin{cases} \text{await } t[\square] \text{ then } \mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(P) & \text{if } lev(V, \vec{t}.t) = \text{Low} \\ \text{stop} & \text{if } lev(V, \vec{t}.t) = \text{High} \end{cases} \\
\mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(\square) &= \square \\
\mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(\text{stop}) &= \text{stop} \\
\mathcal{E}_{V, \mathcal{L}, \vec{t}}^P(\vec{t}_0[P]) &= \begin{cases} \vec{t}_0[\mathcal{E}_{V, \mathcal{L}, (\vec{t}.t_0)}^P(P)] & \text{if } lev(V, \vec{t}.t_0) = \text{Low} \\ \text{stop} & \text{if } lev(V, \vec{t}.t_0) = \text{High} \end{cases}
\end{aligned}$$

Figure 23: Erasure of Agents and Processes

$$\begin{aligned}
\mathcal{E}_{V,\mathcal{L}}^L(\text{true}) &= \text{true} \\
\mathcal{E}_{V,\mathcal{L}}^L(\mathcal{L}_1 \wedge \mathcal{L}_2) &= (\mathcal{E}_{V,\mathcal{L}}^L(\mathcal{L}_1) \wedge \mathcal{E}_{V,\mathcal{L}}^L(\mathcal{L}_2)) \\
\mathcal{E}_{V,\mathcal{L}}^L(k \text{ undone}) &= \begin{cases} \text{true} & \text{if } lev(V,k) = \text{High} \\ k \text{ undone} & \text{if } lev(V,k) = \text{Low} \end{cases} \\
\mathcal{E}_{V,\mathcal{L}}^L(k::\vec{t} \hat{a} \vec{c}) &= \begin{cases} \text{true} & \text{if } lev(V,k) = \text{High} \\ k::\vec{t} \hat{a} \vec{c} & \text{if } lev(V,k) = \text{Low and } \exists k_1.V, \mathcal{L} \vdash k \searrow k_1 \text{ and } lev(V,k_1) = \text{Low} \\ k::\vec{t} \hat{a} \vec{c} & \text{if } lev(V,k) = \text{Low and } \exists k_1.V, \mathcal{L} \vdash k \searrow k_1 \text{ and } lev(V,k_1) = \text{High} \\ & \text{and } \exists k_2.(V, \mathcal{L} \vdash_{\text{High} \rightarrow \text{Low}} k_1 \overset{*}{\rightsquigarrow} k_2 \wedge V, \mathcal{L} \not\vdash tid(V, k_2) \text{ aborted}) \\ \text{true} & \text{if } lev(V,k) = \text{Low and } \exists k_1.V, \mathcal{L} \vdash k \searrow k_1 \text{ and } lev(V,k_1) = \text{High} \\ & \text{and } \nexists k_2.(V, \mathcal{L} \vdash_{\text{High} \rightarrow \text{Low}} k_1 \overset{*}{\rightsquigarrow} k_2 \wedge V, \mathcal{L} \not\vdash tid(V, k_2) \text{ aborted}) \end{cases} \\
\mathcal{E}_{V,\mathcal{L}}^L(k::\vec{t} \check{a} \vec{c}) &= \begin{cases} \text{true} & \text{if } lev(V,k) = \text{High} \\ k::\vec{t} \check{a} \vec{c} & \text{if } lev(V,k) = \text{Low} \end{cases} \\
\mathcal{E}_{V,\mathcal{L}}^L(k_1 \searrow k_2) &= \begin{cases} \text{true} & \text{if } lev(V,k_1) = \text{High or } lev(V,k_2) = \text{High} \\ k_1 \searrow k_2 & \text{if } lev(V,k_1) = \text{Low and } lev(V,k_2) = \text{Low} \end{cases} \\
\mathcal{E}_{V,\mathcal{L}}^L(k_1 \rightsquigarrow k_2) &= \begin{cases} \text{true} & \text{if } lev(V,k_2) = \text{High} \\ k_0 \searrow k_2 & \text{if } lev(V,k_1) = \text{High and } lev(V,k_2) = \text{Low} \\ & \text{and } V, \mathcal{L} \vdash_{\text{Low} \rightarrow \text{High}} k_0 \searrow k_1 \text{ and } V, \mathcal{L} \not\vdash tid(V, k_2) \text{ aborted} \\ \text{true} & \text{if } lev(V,k_1) = \text{High and } lev(V,k_2) = \text{Low} \\ & \text{and } V, \mathcal{L} \vdash_{\text{Low} \rightarrow \text{High}} k_0 \searrow k_1 \text{ and } V, \mathcal{L} \vdash tid(V, k_2) \text{ aborted} \\ k_0 \rightsquigarrow k_2 & \text{if } lev(V,k_1) = \text{High and } lev(V,k_2) = \text{Low} \\ & \text{and } V, \mathcal{L} \vdash_{\text{Low} \rightarrow \text{High}} k_0 \overset{*}{\rightsquigarrow} k_1 \\ k_1 \rightsquigarrow k_2 & \text{if } lev(V,k_1) = \text{Low and } lev(V,k_2) = \text{Low} \end{cases} \\
\mathcal{E}_{V,\mathcal{L}}^L(\vec{t} \square) &= \begin{cases} \text{true} & \text{if } lev(V, \vec{t}) = \text{High} \\ \vec{t} \square & \text{if } lev(V, \vec{t}) = \text{Low} \end{cases} \\
\mathcal{E}_{V,\mathcal{L}}^L(\vec{t} \boxtimes) &= \begin{cases} \text{true} & \text{if } lev(V, \vec{t}) = \text{High} \\ \vec{t} \boxtimes & \text{if } lev(V, \vec{t}) = \text{Low} \end{cases}
\end{aligned}$$

Figure 24: Erasure of Logs

$$\begin{aligned}
\mathcal{U}_{V,\mathcal{L},\mathcal{M}}(\text{true}) &= \text{true} \\
\mathcal{U}_{V,\mathcal{L},\mathcal{M}}(\mathcal{L}_1 \wedge \mathcal{L}_2) &= (\mathcal{U}_{V,\mathcal{L},\mathcal{M}}(\mathcal{L}_1) \wedge \mathcal{U}_{V,\mathcal{L},\mathcal{M}}(\mathcal{L}_2)) \\
\mathcal{U}_{V,\mathcal{L},\mathcal{M}}(k \text{ undone}) &= \text{true} \\
\mathcal{U}_{V,\mathcal{L},\mathcal{M}}(k::\vec{t} \hat{a} \vec{c}) &= \bigwedge \left\{ \begin{array}{l} k_0::\vec{t} \hat{a} \vec{c} \\ \wedge k_0 \searrow k_2 \\ \wedge k_2 \text{ undone} \end{array} \left| \begin{array}{l} \text{lev}(V,k) = \text{Low and } \exists k_1.V,\mathcal{L} \vdash k \searrow k_1 \\ \text{and } \text{lev}(V,k_1) = \text{High and } V,\mathcal{L} \vdash_{\text{High} \rightarrow \text{Low}} k_1 \overset{*}{\rightsquigarrow} k_2 \\ \text{and } V,\mathcal{L} \vdash \text{tid}(V,k_2) \text{ aborted and } k_0 = \mathcal{M}(k,k_2) \end{array} \right. \right\} \\
\mathcal{U}_{V,\mathcal{L},\mathcal{M}}(k::\vec{t} \check{a} \vec{c}) &= \text{true} \\
\mathcal{U}_{V,\mathcal{L},\mathcal{M}}(k_1 \searrow k_2) &= \text{true} \\
\mathcal{U}_{V,\mathcal{L},\mathcal{M}}(k_1 \rightsquigarrow k_2) &= \text{true} \\
\mathcal{U}_{V,\mathcal{L},\mathcal{M}}(\vec{t} \square) &= \text{true} \\
\mathcal{U}_{V,\mathcal{L},\mathcal{M}}(\vec{t} \boxtimes) &= \text{true} \\
\\
\mathcal{R}_{V,\mathcal{L},\mathcal{M}}(\text{true}) &= \text{stop} \\
\mathcal{R}_{V,\mathcal{L},\mathcal{M}}(\mathcal{L}_1 \wedge \mathcal{L}_2) &= (\mathcal{R}_{V,\mathcal{L},\mathcal{M}}(\mathcal{L}_1) \mid \mathcal{R}_{V,\mathcal{L},\mathcal{M}}(\mathcal{L}_2)) \\
\mathcal{R}_{V,\mathcal{L},\mathcal{M}}(k \text{ undone}) &= \text{stop} \\
\mathcal{R}_{V,\mathcal{L},\mathcal{M}}(k::\vec{t} \hat{a} \vec{c}) &= \begin{cases} \vec{t} \hat{a} \vec{c} & \text{if } \text{lev}(V,k) = \text{Low and } \exists k_1.V,\mathcal{L} \vdash k \searrow k_1 \text{ and } \text{lev}(V,k_1) = \text{High} \\ & \text{and } \forall k_2.(V,\mathcal{L} \vdash_{\text{High} \rightarrow \text{Low}} k_1 \overset{*}{\rightsquigarrow} k_2 \Rightarrow V,\mathcal{L} \vdash \text{tid}(V,k_2) \text{ aborted}) \\ \text{true} & \text{otherwise} \end{cases} \\
\mathcal{R}_{V,\mathcal{L},\mathcal{M}}(k::\vec{t} \check{a} \vec{c}) &= \text{stop} \\
\mathcal{R}_{V,\mathcal{L},\mathcal{M}}(k_1 \searrow k_2) &= \text{stop} \\
\mathcal{R}_{V,\mathcal{L},\mathcal{M}}(k_1 \rightsquigarrow k_2) &= \text{stop} \\
\mathcal{R}_{V,\mathcal{L},\mathcal{M}}(\vec{t} \square) &= \text{stop} \\
\mathcal{R}_{V,\mathcal{L},\mathcal{M}}(\vec{t} \boxtimes) &= \text{stop} \\
\\
\frac{k = k_0 \quad k_n = k' \quad \text{lev}(V,k') = \text{Low} \quad V,\mathcal{L} \vdash k_i \rightsquigarrow k_{i+1} \text{ and } \text{lev}(V,k_i) = \text{High for } i = 0, \dots, n-1}{V,\mathcal{L} \vdash_{\text{High} \rightarrow \text{Low}} k \overset{*}{\rightsquigarrow} k'} & \text{(LOGJ HIGHTOLOW)} \\
\frac{k = k_0 \quad k_n = k' \quad \text{lev}(V,k) = \text{Low} \quad V,\mathcal{L} \vdash k_0 \searrow k_1 \text{ and } \text{lev}(V,k_1) = \text{High} \quad V,\mathcal{L} \vdash k_i \rightsquigarrow k_{i+1} \text{ and } \text{lev}(V,k_{i+1}) = \text{High for } i = 1, \dots, n-1}{V,\mathcal{L} \vdash_{\text{Low} \rightarrow \text{High}} k \searrow k'} & \text{(LOGJ LOWTOHIGHSENT)} \\
\frac{k = k_0 \quad k_n = k' \quad \text{lev}(V,k) = \text{Low} \quad V,\mathcal{L} \vdash k_i \rightsquigarrow k_{i+1} \text{ and } \text{lev}(V,k_{i+1}) = \text{High for } i = 0, \dots, n-1}{V,\mathcal{L} \vdash_{\text{Low} \rightarrow \text{High}} k \overset{*}{\rightsquigarrow} k'} & \text{(LOGJ LOWTOHIGHXFER)}
\end{aligned}$$

Figure 25: Auxiliary Functions for Log Erasure

$$\begin{aligned}
\mathcal{E}_{V,\mathcal{L}}^F(\mathcal{F}_1 \wedge \mathcal{F}_2) &= \mathcal{E}_{V,\mathcal{L}}^F(\mathcal{F}_1) \wedge \mathcal{E}_{V,\mathcal{L}}^F(\mathcal{F}_2) \\
\mathcal{E}_{V,\mathcal{L}}^F(\mathcal{F}_1 \vee \mathcal{F}_2) &= \mathcal{E}_{V,\mathcal{L}}^F(\mathcal{F}_1) \vee \mathcal{E}_{V,\mathcal{L}}^F(\mathcal{F}_2) \\
\mathcal{E}_{V,\mathcal{L}}^F(\forall X:T.\mathcal{F}) &= \forall X:T.\mathcal{E}_{V,\mathcal{L}}^F(\mathcal{F}) \\
\mathcal{E}_{V,\mathcal{L}}^F(\exists X:T.\mathcal{F}) &= \exists X:T.\mathcal{E}_{V,\mathcal{L}}^F(\mathcal{F}) \\
\mathcal{E}_{V,\mathcal{L}}^F(\neg \mathcal{F}^A) &= \neg \mathcal{E}_{V,\mathcal{L}}^F(\mathcal{F}^A) \\
\mathcal{E}_{V,\mathcal{L}}^F(\vec{t}_1 \rightsquigarrow \vec{t}_2) &= \begin{cases} \text{true} & \text{if } \text{High} \in \{\text{lev}(V, \vec{t}_1), \text{lev}(V, \vec{t}_2)\} \\ \vec{t}_1 \rightsquigarrow \vec{t}_2 & \text{otherwise} \end{cases} \\
\mathcal{E}_{V,\mathcal{L}}^F(k::\vec{t} \hat{a} \vec{c}) &= \begin{cases} \text{true} & \text{if } \text{lev}(V, k) = \text{High} \\ k::\vec{t} \hat{a} \vec{c} & \text{if } \text{lev}(V, k) = \text{Low} \end{cases} \\
\mathcal{E}_{V,\mathcal{L}}^F(k::\vec{t} \check{a} \vec{c}) &= \begin{cases} \text{true} & \text{if } \text{lev}(V, k) = \text{High} \\ k::\vec{t} \check{a} \vec{c} & \text{if } \text{lev}(V, k) = \text{Low} \end{cases} \\
\mathcal{E}_{V,\mathcal{L}}^F(\vec{t} [\square]) &= \begin{cases} \text{true} & \text{if } \text{lev}(V, \vec{t}) = \text{High} \\ \vec{t} [\square] & \text{otherwise} \end{cases} \\
\mathcal{E}_{V,\mathcal{L}}^F(k \text{ undone}) &= \begin{cases} \text{true} & \text{if } \text{lev}(V, k) = \text{High} \\ k \text{ undone} & \text{otherwise} \end{cases} \\
\mathcal{E}_{V,\mathcal{L}}^F(k_1 = k_2) &= \begin{cases} \text{true} & \text{if } \text{High} \in \{\text{lev}(V, k_1), \text{lev}(V, k_2)\} \\ k_1 = k_2 & \text{otherwise} \end{cases} \\
\mathcal{E}_{V,\mathcal{L}}^F(\text{true}) &= \text{true} \\
\mathcal{E}_{V,\mathcal{L}}^F(\text{false}) &= \text{false}
\end{aligned}$$

Figure 26: Erasure of Constraints

Definition 6.5 (Level of a Reaction) Suppose $V \vdash A \text{ agent}^\ell$ and $(V_1, A_1) \xrightarrow{\mathcal{F}} (V_2, A_2)$. Define the level of the reaction based on the derivation of the reaction:

1. If the last rule in the derivation is (REACT FINISH) for a transaction \vec{t} , then the level of the reaction is $\text{lev}(V, \vec{t})$.
2. If the last rule in the derivation is (REACT AWAIT) for a transaction $\vec{t}.t_0$, then the level of the reaction is $\text{lev}(V, \vec{t}.t_0)$.
3. If the last rule in the derivation is (REACT SYNC), (REACT TRANSFER) or (REACT PREEMPT), for a receiving agent $\vec{t}_1(\check{x} P_1 + P_2)$, then the level of the step is $\text{lev}(V, a)$.
4. If the last rule in the derivation is (REACT NEW), (REACT PAR), (REACT STRUCT) or (REACT EQUIV), then the level of the reaction is that same as that for the antecedent.

Lemma 6.5 (Erased Reactions) Suppose $V \vdash A \text{ agent}^\ell$ and $(V_1, A_1) \xrightarrow{\mathcal{F}} (V_2, A_2)$.

1. If the level of the reaction is high, then $\text{lev}(V_1, \vec{t}) = \text{High}$, then

$$V_1 \vdash \mathcal{E}_{V_1, \mathcal{L}, \mathcal{M}}^A(A_1) \xrightarrow{\mathcal{E}_{V, \mathcal{L}}^F(\mathcal{F})} \mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(A_2).$$

$$\mathcal{E}_{V_1, \mathcal{L}, \mathcal{M}}^A(A_1) \equiv \mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(A_2)$$

2. If the level of the reaction is low, then $\text{lev}(V_1, \vec{t}) = \text{Low}$, then

$$(V_1, \mathcal{E}_{V_1, \mathcal{L}, \mathcal{M}}^A(A_1)) \xrightarrow{\mathcal{E}_{V, \mathcal{L}}^F(\mathcal{F})} (V_2, \mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(A_2)).$$

Lemma 6.6 (Erased Barbs) Suppose V is a low-level environment and $V \vdash A \text{ agent}^\ell$, and $\text{lev}(V, \vec{t}) = \text{Low}$. If $(V, A) \downarrow_{\vec{t}a}^{\mathcal{F}}$, then $(V, \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A)) \downarrow_{\vec{t}a}^{\mathcal{F}'}$, where $\mathcal{F}' \equiv \mathcal{E}_{V, \mathcal{L}}^F(\mathcal{F})$.

PROOF: Suppose $(V, A) \downarrow_{\vec{t}a}^{\mathcal{F}}$, then $V \vdash A \xrightarrow{\mathcal{F}_0} (\mathbf{v} \vec{a} : \vec{C})(\vec{t} \hat{a} \vec{c} \mid A')$ for some A' , where $a \notin \{\vec{a}\}$ and where $\mathcal{F} \equiv \mathcal{F}_0 \wedge \mathcal{F}'_0$ for some \mathcal{F}'_0 . Therefore we have

$$V \vdash \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A) \xrightarrow{\mathcal{E}_{V, \mathcal{L}}^F(\mathcal{F}_0)} \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A((\mathbf{v} \vec{a} : \vec{C})(\vec{t} \hat{a} \vec{c} \mid A')).$$

Since V is low-level and $a \notin \{\vec{a}\}$, the level of a must be low. Therefore:

$$\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A((\mathbf{v} \vec{a} : \vec{C})(\vec{t} \hat{a} \vec{c} \mid A')) = (\mathbf{v} \vec{a} : \vec{C})(\vec{t} \hat{a} \vec{c} \mid \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A')).$$

Therefore we have that $(V, A) \downarrow_{\vec{t}a}^{\mathcal{F}}$. □

We now consider some results for the simulation of an erased agent by the original agent. We wish to verify that the erasure of a process may simulate transitions in the original process. In general the original process may need to perform reaction steps that are not present in evolution of the erased process. The reason is because high processes may hold locks that are required by low processes, and preemptive abort is used to transparently free those locks. Consider the example:

$$A_0 = \hat{a} \mid t_1(\check{a}; P_1) \mid t_2(\check{a}; P_2)$$

where a has type $\text{Lock}(\)^{\text{Low}}$, the transaction t_1 is a high transaction, and t_2 is a low transaction. This may evolve to the agent expression:

$$\begin{aligned} A_1 &= t_1 P_1 \mid t_2(\check{a}; P_2) \mid \llbracket \mathcal{L}_1 \rrbracket \\ \mathcal{L}_1 &= k_0 :: \hat{a} \wedge k_1 :: t_1 \check{a} \wedge k_0 \searrow k_1 \end{aligned}$$

The low transaction t_2 may now preempt the high transaction t_1 , causing this system to evolve to:

$$\begin{aligned} A_2 &= \hat{a} \mid t_2(\check{a}; P_2) \mid \llbracket \mathcal{L}_2 \rrbracket \\ \mathcal{L}_2 &= \mathcal{L}_1 \wedge t_1 \boxtimes \wedge k_1 \text{ undone} \end{aligned}$$

By a synchronization action, this may further evolve to:

$$\begin{aligned} A_3 &= t_2 P_2 \mid \llbracket \mathcal{L}_3 \rrbracket \\ \mathcal{L}_3 &= \mathcal{L}_2 \wedge k_2 :: \hat{a} \wedge k_3 :: t_2 \check{a} \wedge k_2 \searrow k_3 \end{aligned}$$

However, all three of the agent expressions A_0 , A_1 and A_2 have the same erasure:

$$\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_0) = \hat{a} \mid t_2(\check{a}; \mathcal{E}_{V, \mathcal{L}, t_2}^P(P_2))$$

for some V , \mathcal{L} and \mathcal{M} . In particular, the reaction step

$$V \vdash \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_2) \Longrightarrow \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_3)$$

involves the application of the (REACT SYNC) rule in the erased agent expression. The simulation of this step in the original agent expression requires additional applications of (REACT PREEMPT) and (RED UNDO). However since these additional reaction steps are driven by the attempt by a low process to acquire a lock, this lock acquisition step is also present in the erased process, and therefore consideration of this sequence of preemption and undoing operations can be deferred until we consider simulating a lock acquisition or lock transfer in the erasure of a process.

With these explanatory remarks, we first consider only reductions. We are required to verify the dashed arrows in this figure:

$$\begin{array}{ccc} A_1 & \xrightarrow{\mathcal{F}_2} & \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A) & A \\ \mathcal{F}_1 \downarrow & & & \mathcal{F}_1 \downarrow \\ A'_1 & \dashrightarrow^{\mathcal{F}_2} & \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A') & A' \end{array}$$

The eventual goal is to verify that the erasure of a low agent is bisimilar to the original agent. We need to strengthen the coinduction hypothesis in the definition of the bisimulation, to relate not the erasure of an agent back to that agent, but to relate a process that may evolve to that erasure. So there may be a process in the simulation A_1 that does not correspond exactly to the erasure of a process, but it can at least transition by reduction to such an erasure $\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A)$. The complication is there may be an alternative reduction to a process A'_1 . So we need to verify that, if the latter reduction is taken, then it is still possible in the original unerased agent to simulate this reduction and evolve via further reductions to a process that is the erasure $\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A')$ of an original process A' .

The only reductions that affect the logs are applications of the (RED UNDO) rule, and the only log entries added by these reduction steps that may affect the applicability of other reduction steps are of the

form (k undone). It will be useful to have the notion of the removal from a constraint set of the constraints that require that certain message receives have *not* been undone:

$$\begin{aligned} \text{entries}(\mathcal{F}) &= \begin{cases} \{\} & \text{if } \mathcal{F} = \text{true} \\ \text{entries}(\mathcal{F}_1) \cup \text{entries}(\mathcal{F}_2) & \text{if } \mathcal{F} = (\mathcal{F}_1 \wedge \mathcal{F}_2) \\ \{\mathcal{F}\} & \text{otherwise} \end{cases} \\ \text{undos}(\mathcal{F}) &= \{\mathcal{F}' \in \text{entries}(\mathcal{F}) \mid \exists k. \mathcal{F}' = \neg(k \text{ undone})\} \\ \mathcal{F}_1 \setminus \mathcal{F}_2 &= \bigwedge \{\mathcal{F}' \in \text{entries}(\mathcal{F}_1) \mid \mathcal{F}' \notin \text{entries}(\mathcal{F}_2)\} \end{aligned}$$

To verify the result, we verify the following strengthened hypothesis:

Lemma 6.7 (Simulation of Reductions) *Suppose*

$$\begin{aligned} &V \vdash A \text{ agent}^\ell \\ &V \vdash A_1 \xrightarrow{\mathcal{F}_1} A'_1 \\ &V \vdash (A_1 \mid A_2) \xrightarrow{\mathcal{F}_2} \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A) \\ &V, \text{true} \vdash \mathcal{C}[A_1 \mid A_2] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}_1 \wedge \mathcal{F}_2] \\ &\mathcal{L} = (\mathcal{C}'[A])^* \text{ and } \mathcal{C} = \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(\mathcal{C}') \end{aligned}$$

Then there exists A' and \mathcal{F}'_1 such that

$$\begin{aligned} &V \vdash A \xrightarrow{\mathcal{F}_1} A' \\ &V \vdash (A'_1 \mid A_2) \xrightarrow{\mathcal{F}_2} \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A') \\ &V, \text{true} \vdash \mathcal{C}[A'_1 \mid A_2] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}'_1] \\ &V, \text{true} \vdash \mathcal{C}'[A] \triangleright \mathcal{F} \supset \mathcal{C}'[\mathcal{F}'_1] \\ &\mathcal{F}'_1 = \mathcal{F}_1 \setminus \text{undos}(\mathcal{F}_2) \text{ and } \mathcal{F}'_2 = \mathcal{F}_2 \setminus \text{undos}(\mathcal{F}_1) \end{aligned}$$

We are required to verify the dashed arrows in this figure:

$$\begin{array}{ccccc} A_1 & (A_1 \mid A_2) & \xrightarrow{\mathcal{F}_2} & \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A) & A \\ \mathcal{F}_1 \downarrow & \downarrow & & & \mathcal{F}'_1 \downarrow \\ A'_1 & (A'_1 \mid A_2) & \xrightarrow{\mathcal{F}'_2} & \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A') & A' \end{array}$$

The complication in the statement of the lemma is the preconditions on the application of the log rules. We are concerned with a scenario where two different reduction rules are applicable in an agent expression $(A_1 \mid A_2)$. The constraints are \mathcal{F}_1 and \mathcal{F}_2 , respectively. Since the constraints are intended to restrict the possible contexts within which this computation may take place, we verify the lemma in an arbitrary context \mathcal{C} , corresponding to the erasure of a context \mathcal{C}' . The constraint \mathcal{F} corresponds to the constraint on this global agent expression $\mathcal{C}'[A]$. This global constraint must entail both preconditions \mathcal{F}_1 and \mathcal{F}_2 , with the log entries in $(A_1 \mid A_2)$, combined with log entries in the global context \mathcal{C} . Note that we must require that the precondition \mathcal{F}_1 is consistent with the enlarged context $(A_1 \mid A_2)$, rather than simply A_1 , since it is in the former context that we see the possibility of two different reduction steps being available. The erasure function is indexed by a global log \mathcal{L} that can be extracted from the global agent expression $\mathcal{C}'[A]$. The

lemma verifies that the above diagram commutes, and in particular that the reductions from $(A'_1 \mid A_2)$ to $\mathcal{E}_{V,\mathcal{L},\mathcal{M}}^A(A')$, and from A to A' , are enabled by their respective contexts.

PROOF: We perform the verification by induction on a derivation of $V \vdash A_1 \xrightarrow{\mathcal{F}_1} A'_1$. We consider the possible cases for a reduction step:

1. (RED IFTRUE): $A_1 = \vec{t}((a=a) \rightarrow P_1 \parallel P_2)$ and the reduct is $A'_1 = \vec{t}P_1$.
 - (a) Suppose that $V \vdash (A_1 \mid A_2) \xrightarrow{\mathcal{F}_2} \mathcal{E}_{V,\mathcal{L},\mathcal{M}}^A(A)$ is derived from $V \vdash A_1 \xRightarrow{A'_1}$. Then $\mathcal{F}'_1 = \mathcal{F}'_2 = \text{true}$ and let $A' = A$.
 - (b) Otherwise the two reduction steps result from different redices. So $V \vdash ([\] \mid A_2) \xrightarrow{\mathcal{F}_2} \mathcal{E}_{V,\mathcal{L},\mathcal{M}}^A(\mathcal{C}_0)$, for some context \mathcal{C}_0 where $\mathcal{E}_{V,\mathcal{L},\mathcal{M}}^A(\mathcal{C}_0)[A_1] = \mathcal{E}_{V,\mathcal{L},\mathcal{M}}^A(A)$. So we have $A = \mathcal{C}_0[(a=a) \rightarrow P'_1 \parallel P'_2]$ where $\mathcal{E}_{V,\mathcal{L},\mathcal{M}}^A(P'_1) = P_1$ and $\mathcal{E}_{V,\mathcal{L},\mathcal{M}}^A(P'_2) = P_2$. Let $A' = \mathcal{C}_0[P'_1]$ and $\mathcal{F}'_1 = \text{true}$ and $\mathcal{F}'_2 = \mathcal{F}_2$, then $V \vdash A \xRightarrow{A'}$.
2. (RED UNDO): Then $V \vdash \text{stop} \xrightarrow{\mathcal{F}_1} (\vec{t}_2 \hat{a} \vec{c} \mid \llbracket k_1 \text{ undone} \rrbracket)$ where $\mathcal{F}_1 = \mathcal{F}(k_1 :: \vec{t}_1 \check{a} \vec{c} \text{ undoable}) \wedge k_2 \searrow k_1 \wedge k_2 :: \vec{t}_2 \hat{a} \vec{c}$.
 - (a) Suppose that $V \vdash (A_1 \mid A_2) \xrightarrow{\mathcal{F}_2} \mathcal{E}_{V,\mathcal{L},\mathcal{M}}^A(A)$ is derived from $V \vdash A_1 \xRightarrow{A'_1}$. This ensures that $V, \mathcal{C}^* \vdash (A_1 \mid A_2) \triangleright \mathcal{F}_2 \supset ([\mathcal{F}_1] \mid A_2)$. This corresponds to the \mathcal{F}_2 reduction step being a sequence of undo steps that includes the undo step in \mathcal{F}_1 , therefore it is neither necessary nor possible to redo this undo step from $\mathcal{E}_{V,\mathcal{L},\mathcal{M}}^A(A)$. Let $A' = A$. Let \mathcal{F}'_1 and \mathcal{F}'_2 be defined as in the statement of the lemma. In this case, $\text{undos}(\mathcal{F}_1)$ is a singleton $\{\neg(k_1 \text{ undone})\}$, so $\text{undos}(\mathcal{F}'_1) = \text{undos}(\mathcal{F}_1 \setminus \text{undos}(\mathcal{F}_2)) = \text{true}$, while \mathcal{F}'_2 is the result of removing the single constraint $\neg(k_1 \text{ undone})$ from \mathcal{F}_2 . We need to verify the two conditions:

$$\begin{aligned} V, \text{true} \vdash \mathcal{C}[A'_1 \mid A_2] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}'_2] \\ V, \text{true} \vdash \mathcal{C}'[A] \triangleright \mathcal{F} \supset \mathcal{C}'[\mathcal{F}'_1] \end{aligned}$$

The latter of these follows trivially. The former follows from the preconditions of the lemma and the fact that $(A'_1)^* = A_1^* \wedge (k_1 \text{ undone})$.

- (b) Otherwise the two reduction steps result from different redices. So again $V \vdash ([\] \mid A_2) \xrightarrow{\mathcal{F}_2} \mathcal{E}_{V,\mathcal{L},\mathcal{M}}^A(\mathcal{C}_0)$, for some context \mathcal{C}_0 where $\mathcal{E}_{V,\mathcal{L},\mathcal{M}}^A(\mathcal{C}_0)[A_1] = \mathcal{E}_{V,\mathcal{L},\mathcal{M}}^A(A)$. So $A = \mathcal{C}_0[\text{stop}]$ and we need to verify that $V \vdash A \xrightarrow{\mathcal{F}'_1} \mathcal{C}_0[\vec{t}_2 \hat{a} \vec{c} \mid \llbracket k_1 \text{ undone} \rrbracket]$ is justified by the context \mathcal{C}' . Since \mathcal{C}_0 is the result of the reduction steps from $(A_1 \mid A_2)$ with the context constraint \mathcal{F}_2 , we have that

$$\begin{aligned} (\mathcal{C}_0[\text{stop}])^* \equiv & A_2^* \wedge \bigwedge \{(k \text{ undone}) \mid \\ & (\neg(k \text{ undone})) \in \text{undos}(\mathcal{F}_2)\} \end{aligned}$$

There are two cases to consider:

- i. $(\neg(k_1 \text{ undone})) \in \text{undos}(\mathcal{F}_2)$: So the message receipt that is undone by the reduction with constraint \mathcal{F}_1 is also among those done by the reduction with constraint \mathcal{F}_2 . This case is similar to the case where the reduction with constraint \mathcal{F}_2 is derived from the reduction with constraint \mathcal{F}_1 . The only difference is that the global nature of the logs allows the same reduction to be performed at different points in the context. The reasoning is as before.

ii. $(\neg(k_1 \text{ undone})) \notin \text{undos}(\mathcal{F}_2)$: We are given

$$V, \text{true} \vdash \mathcal{C}[\text{stop} \mid A_2] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}_1]$$

So by the fact that $\mathcal{C}_0[\text{stop}]$ only adds undo log entries for events different from k_1 , and the reduction with the \mathcal{F}_1 constraint only introduces an undo log entry for k_1 , we have that

$$V, \text{true} \vdash \mathcal{C}[\mathcal{C}_0[\text{stop}]] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}_1]$$

i.e., the constraint $\neg(k_1 \text{ undone})$ is consistent with the log entries in $\mathcal{C}_0[\text{stop}]$. So we can conclude that

$$V, \text{true} \vdash \mathcal{C}[A] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}_1].$$

We are also given that

$$V, \text{true} \vdash \mathcal{C}[\text{stop} \mid A_2] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}_2].$$

Since \mathcal{F}_2 only enforces the absence of undo log entries for events different from k_1 , we have that

$$V, \text{true} \vdash \mathcal{C}[[k_1 \text{ undone}] \mid A_2] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}_2].$$

Since $(A'_1)^* = (k_1 \text{ undone})$, we conclude that

$$V, \text{true} \vdash \mathcal{C}[A'_1 \mid A_2] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}_2].$$

□

Corollary 6.1 *Suppose*

$$\begin{aligned} & V \vdash A \text{ agent}^\ell \\ & V \vdash A_1 \xrightarrow{\mathcal{F}_1} A'_1 \\ & V \vdash A_1 \xrightarrow{\mathcal{F}_2} \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A) \\ & V, \text{true} \vdash \mathcal{C}[A_1] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}_1 \wedge \mathcal{F}_2] \\ & \mathcal{L} = (\mathcal{C}'[A])^* \text{ and } \mathcal{C} = \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(\mathcal{C}') \end{aligned}$$

Then there exists A' and \mathcal{F}'_1 such that

$$\begin{aligned} & V \vdash A \xrightarrow{\mathcal{F}_1} A' \\ & V \vdash A'_1 \xrightarrow{\mathcal{F}_2} \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A') \\ & V, \text{true} \vdash \mathcal{C}[A'_1] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}'_2] \\ & V, \text{true} \vdash \mathcal{C}'[A] \triangleright \mathcal{F} \supset \mathcal{C}'[\mathcal{F}'_1] \\ & \mathcal{F}'_1 = \mathcal{F}_1 \setminus \text{undos}(\mathcal{F}_2) \text{ and } \mathcal{F}'_2 = \mathcal{F}_2 \setminus \text{undos}(\mathcal{F}_1) \end{aligned}$$

PROOF: Set $A_2 = \text{stop}$ in the previous lemma. □

We now consider the case for reactions. We are required to verify the dashed arrows in this figure:

$$\begin{array}{ccccc}
(V_1, A'_1) & & A'_1 & \xrightarrow{\mathcal{F}_2} & \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_1) & (V_1, A_1) \\
\downarrow \vec{t}; \mathcal{F}_1 & & & & & \downarrow \vec{t}; \mathcal{F}'_1 \\
(V_2, A'_2) & & A'_2 & \dashrightarrow^{\mathcal{F}_2} & \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_2) & (V_2, A_2)
\end{array}$$

In other words, a process that is simulating an unerased process may be able to transition in zero or more reduction steps to the erasure of a process. But there may be an alternative reaction step that can be taken to proceed from A'_1 to A'_2 . We must verify that in the resulting process A_2 , it is possible to perform the internal reduction steps that take A'_2 to the erasure of a process.

Lemma 6.8 (Simulation of Reactions) *Suppose*

$$\begin{aligned}
& V_1 \vdash A_1 \text{ agent}^\ell \\
& V_1 \vdash A'_1 \xrightarrow{\mathcal{F}_2} \mathcal{E}_{V_1, \mathcal{L}, \mathcal{M}}^A(A_1) \\
& (V_1, A'_1) \xrightarrow{\mathcal{F}_1} (V_2, A'_2) \\
& V, \text{true} \vdash \mathcal{C}[A'_1] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}_1 \wedge \mathcal{F}_2] \\
& \mathcal{L} = (\mathcal{C}'[A_1])^* \text{ and } \mathcal{C} = \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(\mathcal{C}')
\end{aligned}$$

Then there exists A'_2 and \mathcal{F}'_1 such that

$$\begin{aligned}
& (V_1, A_1) \xrightarrow{* \mathcal{F}'_1} (V_2, A_2) \\
& V_2 \vdash A'_2 \xrightarrow{\mathcal{F}_2} \mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(A_2) \\
& V, \text{true} \vdash \mathcal{C}[A'_2] \triangleright \mathcal{F} \supset \mathcal{C}[\mathcal{F}'_2] \\
& V, \text{true} \vdash \mathcal{C}'[A_1] \triangleright \mathcal{F} \supset \mathcal{C}'[\mathcal{F}'_1] \\
& \mathcal{F}'_1 = \mathcal{F}_1 \setminus \text{undos}(\mathcal{F}_2) \text{ and } \mathcal{F}'_2 = \mathcal{F}_2 \setminus \text{undos}(\mathcal{F}_1)
\end{aligned}$$

PROOF: We verify this by induction on the derivation for $(V_1, A'_1) \xrightarrow{\mathcal{F}_1} (V_2, A'_2)$.

1. Case (REACT SYNC): By Lemma 6.8, we have

$$\begin{aligned}
V_1 \vdash A'_1 & \implies (v \vec{a} : \vec{C})(\vec{t}_2 \hat{a} \vec{c} \mid \vec{t}_1(\vec{a} \vec{x} P_1 + P_2) \mid A''_1) \quad (1) \\
V_2 \vdash (v \vec{a} : \vec{C})(\vec{t}_1 \{ \vec{c} / \vec{x} \} P_1 \mid \llbracket \mathcal{L} \rrbracket \mid A''_1) & \implies A'_2 \quad (2)
\end{aligned}$$

By Corollary 6.1 applied to (1), there is some A_3 and $\mathcal{F}'_{1,1}$ such that

$$\begin{aligned}
& V_1 \vdash A_1 \xrightarrow{\mathcal{F}'_{1,1}} A_3 \\
& V_1 \vdash (v \vec{a} : \vec{C})(\vec{t}_2 \hat{a} \vec{c} \mid \vec{t}_1(\vec{a} \vec{x} P_1 + P_2) \mid A''_1) \xrightarrow{\mathcal{F}_2} \mathcal{E}_{V_1, \mathcal{L}, \mathcal{M}}^A(A_3) \\
& \mathcal{E}_{V_1, \mathcal{L}}^F(\mathcal{F}'_{1,1}) \equiv \text{true}
\end{aligned}$$

Depicted graphically, we have

$$\begin{array}{ccccc}
A'_1 & \xrightarrow{\mathcal{F}_2} & \mathcal{E}_{V_1, \mathcal{L}, \mathcal{M}}^A(A) & & A_1 \\
\downarrow & & & & \vdots \\
& & & & \mathcal{F}'_{1,1} \\
& & & & \vdots \\
(v \vec{a} : \vec{C})(\dots \mid A'_{1,1}) & \dashrightarrow^{\mathcal{F}_2} & \mathcal{E}_{V_1, \mathcal{L}, \mathcal{M}}^A(A_3) & & A_3
\end{array}$$

So we have $\mathcal{E}_{V_1, \mathcal{L}, \mathcal{M}}^A(A_2) = \mathcal{E}_{V_1, \mathcal{L}, \mathcal{M}}^A(\mathcal{C})[\vec{t}_2 \hat{a} \vec{c} \mid \vec{t}_1(\check{a} \vec{x} P_1 + P_2)]$ for some \mathcal{C} , some P'_1 and P'_2 where $\mathcal{E}_{V_1, \mathcal{L}, \vec{t}_1}^P(P'_1) = P_1$ and $\mathcal{E}_{V_1, \mathcal{L}, \vec{t}_1}^P(P'_2) = P_2$. So $A_3 = \mathcal{C}[\vec{t}_2 \hat{a} \vec{c} \mid \vec{t}_1(\check{a} \vec{x} P'_1 + P'_2)]$, and

$$(V_1, A_3) \xrightarrow{\mathcal{F}'_1} (V_2, \mathcal{C}[\vec{t}_1\{\vec{c}/\vec{x}\}P'_1 \mid \llbracket \mathcal{L}'_1 \rrbracket])$$

for some \mathcal{L}'_1 such that $\mathcal{E}_{V_2, \mathcal{L}}^L(\mathcal{L}'_1) = \mathcal{L}_1$. We have

$$V_2 \vdash \vec{t}_1\{\vec{c}/\vec{x}\}P'_1 \mid \llbracket \mathcal{L}'_1 \rrbracket \mid A'_{1,1} \xrightarrow{\mathcal{F}_2} \mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(\mathcal{C})[\vec{t}_1\{\vec{c}/\vec{x}\}P'_1 \mid \llbracket \mathcal{L}'_1 \rrbracket]$$

and

$$\mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(\mathcal{C}[\vec{t}_1\{\vec{c}/\vec{x}\}P'_1 \mid \llbracket \mathcal{L}'_1 \rrbracket]) = \vec{t}_1\{\vec{c}/\vec{x}\}P_1 \mid \llbracket \mathcal{L}_1 \rrbracket.$$

By Corollary 6.1 applied to (2) above, there is some A_4 and $\mathcal{F}'_{1,2}$ such that

$$\begin{aligned} V_2 \vdash \mathcal{C}[\vec{t}_1\{\vec{c}/\vec{x}\}P'_1 \mid \llbracket \mathcal{L}'_1 \rrbracket] &\xrightarrow{\mathcal{F}'_{1,2}} A_4 \\ V_2 \vdash A'_2 &\xrightarrow{\mathcal{F}_2} \mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(\mathcal{C}[\vec{t}_1\{\vec{c}/\vec{x}\}P'_1 \mid \llbracket \mathcal{L}'_1 \rrbracket]) \\ &\mathcal{E}_{V_2, \mathcal{L}}^F(\mathcal{L}'_{1,2}) = \text{true} \end{aligned}$$

Depicted graphically, we have

$$\begin{array}{ccc} (\mathbf{v} \vec{a} : \vec{C})(\dots \mid A'_{1,1}) & \xrightarrow{\mathcal{F}_2} & \mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(\mathcal{C}[\dots]) & \mathcal{C}[\dots] \\ \downarrow & & & \vdots \\ & & & \mathcal{F}'_{1,2} \vdots \\ & & & \downarrow \\ A'_2 & \text{-----} \mathcal{F}_2 \Rightarrow & \mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(A_4) & A_4 \end{array}$$

Composing these three steps, we have the verification of the lemma for this case:

$$\begin{array}{ccc} A'_1 & \xrightarrow{\mathcal{F}_2} & \mathcal{E}_{V_1, \mathcal{L}, \mathcal{M}}^A(A) & A_1 \\ \downarrow & & & \vdots \\ (\mathbf{v} \vec{a} : \vec{C})(\dots \mid A'_{1,1}) & & & \mathcal{F}'_{1,1} \vdots \\ \downarrow & & & \downarrow \\ \mathcal{F}_1 \downarrow & & & A_3 \\ (\mathbf{v} \vec{a} : \vec{C})(\dots \mid A'_{1,1}) & & & \mathcal{F}'_1 \vdots \\ \downarrow & & & \downarrow \\ & & & \mathcal{C}[\dots] \\ \downarrow & & & \vdots \\ A'_2 & \text{-----} \mathcal{F}_2 \Rightarrow & \mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(A_4) & A_4 \end{array}$$

□

Finally the following lemma states that if a barb is offered in the erasure of a process, then it can also be offered in the original process.

Lemma 6.9 (Simulation of Barbs) *If $V \vdash A$ agent^l and $(V, \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A)) \downarrow_{\vec{t}a}^{\mathcal{F}}$, then $(V, A) \Downarrow_{\vec{t}a}^{\mathcal{F}'}$, for some \mathcal{F}' where $\mathcal{E}_{V, \mathcal{L}}^{\mathcal{F}'}(\mathcal{F}') \equiv \mathcal{F}$.*

To verify this lemma, we will find the following notions useful, of agent and process contexts indexed by the transaction in which the context is instantiated:

$$\begin{aligned} \mathcal{C}\mathcal{A}_{\vec{t}} &::= (\mathcal{C}\mathcal{A}_{\vec{t}} \mid A) \mid (A \mid \mathcal{C}\mathcal{A}_{\vec{t}}) \mid (\text{va}:C)\mathcal{C}\mathcal{A}_{\vec{t}} \mid \vec{t}_1 \mathcal{C}\mathcal{P}_{\vec{t}_2} \\ \mathcal{C}\mathcal{P}_{\vec{t}} &::= [] \mid (\mathcal{C}\mathcal{P}_{\vec{t}} \mid P) \mid (P \mid \mathcal{C}\mathcal{P}_{\vec{t}}) \mid \\ &(\text{va}:C)\mathcal{C}\mathcal{P}_{\vec{t}} \mid \vec{t}_1[\mathcal{C}\mathcal{P}_{\vec{t}_2}] \mid \text{repl } \mathcal{C}\mathcal{P}_{\vec{t}} \mid \\ &(v_1=v_2) \rightarrow \mathcal{C}\mathcal{P}_{\vec{t}}[P] \mid (v_1=v_2) \rightarrow P[\mathcal{C}\mathcal{P}_{\vec{t}}] \end{aligned}$$

In the cases for $\vec{t}_1 \mathcal{C}\mathcal{P}_{\vec{t}_2}$ and $\vec{t}_1[\mathcal{C}\mathcal{P}_{\vec{t}_2}]$, we require $\vec{t} = \vec{t}_1.\vec{t}_2$.

PROOF: If $(V, \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A)) \downarrow_{\vec{t}a}^{\mathcal{F}}$, then $V \vdash \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A) \xrightarrow{\mathcal{F}} \mathcal{C}[\vec{t} \hat{a} \vec{c}]$ for some context \mathcal{C} , where a is not bound in \mathcal{C} . We must have $\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A) = \mathcal{C}\mathcal{A}_{\vec{t}}[\hat{a} \vec{c}]$ for some $\mathcal{C}\mathcal{A}_{\vec{t}}$. By the definition of erasure, we have $\mathcal{C}\mathcal{A}_{\vec{t}} \equiv \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(\mathcal{C}\mathcal{A}'_{\vec{t}})$ and $\hat{a} \vec{c} \equiv \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_0)$, for some $\mathcal{C}\mathcal{A}'_{\vec{t}}$ and A_0 such that $A = \mathcal{C}\mathcal{A}'_{\vec{t}}[A_0]$. We have two cases to consider:

1. $A_0 = \hat{a} \vec{c}$: The result follows immediately, since $V \vdash \mathcal{C}\mathcal{A}'_{\vec{t}}[\hat{a} \vec{c}] \Longrightarrow \mathcal{C}'[\vec{t} \hat{a} \vec{c}]$ for some \mathcal{C}' such that $\mathcal{C} \equiv \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(\mathcal{C}')$.
2. $A_0 = [\mathcal{L}_0]$, so the message arises from the extraction of messages from the logs of high processes that are erased, and $\mathcal{L}_0 = \mathcal{L}'_0 \wedge k :: \vec{t} \check{a} \vec{c}$ for some \mathcal{L}'_0 , and for some \mathcal{C}_0 :

$$\begin{aligned} \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A([\mathcal{L}_0]) &= \mathcal{C}_0[\mathcal{R}_{V, \mathcal{L}, \mathcal{M}}(k :: \vec{t} \check{a} \vec{c})] \\ \mathcal{R}_{V, \mathcal{L}, \mathcal{M}}(k :: \vec{t} \check{a} \vec{c}) &= \begin{cases} \vec{t} \hat{a} \vec{c} & \text{if } \text{lev}(V, k) = \text{Low} \text{ and } \exists k_1.V, \mathcal{L} \vdash k \searrow k_1 \\ & \text{and } \text{lev}(V, k_1) = \text{High} \\ & \text{and } \forall k_2.(V, \mathcal{L} \vdash_{\text{High} \rightarrow \text{Low}} k_1 \overset{*}{\curvearrowright} k_2 \\ & \quad \Rightarrow V, \mathcal{L} \vdash \text{tid}(V, k_2) \text{ aborted}) \\ \text{true} & \text{otherwise} \end{cases} \end{aligned}$$

In this case, the second condition says that any low transactions that have subsequently obtained the lock have aborted, so the lock can still be considered to be held by the high transaction \vec{t} . Therefore we can choose \mathcal{F}' to contain either $\vec{t} \square$ or $\vec{t} \boxtimes$, whichever is compatible with the context in the unerased system. The choice of constraining the high transaction to be committed or aborted does not affect transitions in the erased system, since $\mathcal{E}_{V, \mathcal{L}}^L(\vec{t} \square) = \text{true} = \mathcal{E}_{V, \mathcal{L}}^L(\vec{t} \boxtimes)$.

□

The following theorem uses these results to provide a critical result for the erasure of a process:

Theorem 6.1 (Erasure of Low Agent) *If V is a low environment and $V \vdash A$ agent^{Low}, then $\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A) \overset{\bullet}{\approx} A$.*

PROOF: Define R to be the set

$$\left\{ (A_1, A'_1) \left| \begin{array}{l} V \vdash A_1 \text{ agent}^\ell \text{ for some low-level } V \\ \text{and } V \vdash A'_1 \xrightarrow{\mathcal{F}} \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_1) \end{array} \right. \right\}$$

We verify that R defines a bisimilarity. Suppose that $(A_1, A'_1) \in R$, so $V \vdash A_1 \text{ agent}^\ell$ for some low-level V and $V \vdash A'_1 \xrightarrow{\mathcal{F}} \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_1)$.

1. Suppose $(V_1, A_1) \xrightarrow{\mathcal{F}_0} (V_2, A_2)$ for some A_2 . By Theorem 5.1, we have $V \vdash A_2 \text{ agent}^\ell$. By Lemma 6.5, there are two cases to consider, based on the level of the original reaction:

- (a) If the reaction level is low, then $(V_1, \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_1)) \xrightarrow{\mathcal{E}_{V, \mathcal{L}}^F(\mathcal{F}_0)} (V_2, \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_2))$. Define $A'_2 = \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_2)$, so $(V_1, A'_1) \xrightarrow{\mathcal{E}_{V, \mathcal{L}}^F(\mathcal{F}_0)} (V_2, \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_2))$.
- (b) If the reaction level is high, then $V_1 \vdash \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_1) \xrightarrow{\mathcal{E}_{V, \mathcal{L}}^F(\mathcal{F}_0)} \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_2)$. Define $A'_2 = A'_1$, then $V_1 \vdash \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_1) \xrightarrow{\mathcal{F} \wedge \mathcal{E}_{V, \mathcal{L}}^F(\mathcal{F}_0)} \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_2)$.

In either case, we can conclude that $(A_2, A'_2) \in R$.

2. Suppose $(V_1, A'_1) \xrightarrow{\mathcal{F}_0} (V_2, A'_2)$ for some A'_2 . Then by Lemma 6.8 there is some A_2 and \mathcal{F}'_0 such that

$$\begin{array}{l} (V_1, A_1) \xrightarrow{*\mathcal{F}'_0} (V_2, A_2) \\ V_2 \vdash A_2 \xrightarrow{\mathcal{F}} \mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(A_2) \\ \mathcal{E}_{V_2, \mathcal{L}}^F(\mathcal{F}'_0) \equiv \mathcal{F}_0. \end{array}$$

By Theorem 5.1, we have $V_2 \vdash A_2 \text{ agent}^\ell$. So we can conclude that $(A_2, A'_2) \in R$.

3. Finally we consider barbs. Suppose $(V_1, A_1) \downarrow_{\overline{t}a}^{\mathcal{F}_1}$, so we have $V \vdash A_1 \xrightarrow{\mathcal{F}_1} A_2$ and $(V, A_2) \downarrow_{\overline{t}a}^{\mathcal{F}_1}$. Then by Lemma 6.6 we have $(V_1, \mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A_2)) \downarrow_{\overline{t}a}^{\mathcal{F}_1}$, where $\mathcal{F} \equiv \mathcal{E}_{V_1, \mathcal{L}}^F(\mathcal{F}'_1)$. We have that $(V_1, A'_1) \xrightarrow{*\mathcal{F}_2} (V_2, A'_2)$ by the first property that has been verified for the R relation, for some V_2 and \mathcal{F}_2 , where we have $V_2 \vdash A'_2 \xrightarrow{\mathcal{F}_3} \mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(A_2)$. So we have $(V_2, A'_2) \downarrow_{\overline{t}a}^{\mathcal{F}_1 \wedge \mathcal{F}_3}$, so we conclude that $(V_1, A'_1) \downarrow_{\overline{t}a}^{\mathcal{F}_1 \wedge \mathcal{F}_2 \wedge \mathcal{F}_3}$.

Suppose on the other hand that $(V_1, A'_1) \downarrow_{\overline{t}a}^{\mathcal{F}_1}$, so we have $V_1 \vdash A'_1 \xrightarrow{\mathcal{F}_1} A'_2$ and $(V_1, A'_2) \downarrow_{\overline{t}a}^{\mathcal{F}_1}$. By the second property that has been verified for the R relation, we have that $(V_1, A_1) \xrightarrow{*\mathcal{F}_2} (V_2, A_2)$ for some V_2 and \mathcal{F}_2 , where $V_2 \vdash A_2 \text{ agent}^\ell$ and $V_2 \vdash A'_2 \xrightarrow{\mathcal{F}_2} \mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(A_2)$. By Lemma 6.1, if $(V_2, A'_2) \downarrow_{\overline{t}a}^{\mathcal{F}}$ then $(V_2, \mathcal{E}_{V_2, \mathcal{L}, \mathcal{M}}^A(A'_2)) \downarrow_{\overline{t}a}^{\mathcal{F}}$. Then by Lemma 6.9 we have $(V_2, A'_2) \downarrow_{\overline{t}a}^{\mathcal{F}'_1}$ where $\mathcal{E}_{V_2, \mathcal{L}}^F(\mathcal{F}'_1) \equiv \mathcal{F}$.

□

Lemma 6.10 (Erasure of High Agent) *If $V \vdash A \text{ agent}^{\text{High}}$, then $\mathcal{E}_{V, \mathcal{L}, \mathcal{M}}^A(A) \equiv \text{stop}$.*

PROOF: By induction on the structure of A . □

We define $low(V)$ to be result of replacing all security level annotations in types in V with Low. We define $low(\mathcal{C})$ to be the result of replacing all security level annotations in types in \mathcal{C} with Low.

Lemma 6.11 *Assume V is a low-level type environment. If \mathcal{C} is a (V, ℓ) - V', ℓ' context, then $low(\mathcal{C})$ is a (V, ℓ) - $(low(V'), Low)$ context.*

PROOF: By a straightforward induction on the derivation of $V' \vdash \mathcal{C} \text{ agent}^{\ell'}$. □

Theorem 6.2 (Noninterference) *Suppose V is a low-level environment, and \mathcal{C}_0 is a $(V_0, High)$ - (V, ℓ) environment. If $V_0 \vdash A_1 \text{ agent}^{High}$ and $V_0 \vdash A_2 \text{ agent}^{High}$, then $\mathcal{C}_0[A_1] \approx_{V, \ell} \mathcal{C}_0[A_2]$.*

PROOF: Suppose V' is closed and \mathcal{C} is a (V, ℓ) - (V', ℓ') context. Define $\mathcal{C}' = low(\mathcal{C})$, then \mathcal{C}' is a (V, ℓ) - $(low(V'), Low)$ context. Then we have:

$$\begin{aligned}
\mathcal{C}'[\mathcal{C}_0[A_1]] &\stackrel{\bullet}{\approx} \mathcal{E}_{V_0, \mathcal{L}, \mathcal{M}}^A(\mathcal{C}'[\mathcal{C}_0[A_1]]) \text{ by Lemma 6.2} \\
&= \mathcal{E}_{low(V'), \mathcal{L}, \mathcal{M}}^A(\mathcal{C}'[\mathcal{C}_0])[\mathcal{E}_{V_0, \mathcal{L}, \mathcal{M}}^A(A_1)] \text{ by Theorem 6.1} \\
&\equiv \mathcal{E}_{low(V'), \mathcal{L}, \mathcal{M}}^A(\mathcal{C}'[\mathcal{C}_0])[\text{stop}] \text{ by Lemma 6.10} \\
&\equiv \mathcal{E}_{low(V'), \mathcal{L}, \mathcal{M}}^A(\mathcal{C}'[\mathcal{C}_0])[\mathcal{E}_{V_0, \mathcal{L}, \mathcal{M}}^A(A_2)] \text{ by Lemma 6.10} \\
&= \mathcal{E}_{V_0, \mathcal{L}, \mathcal{M}}^A(\mathcal{C}'[\mathcal{C}_0[A_2]]) \text{ by Theorem 6.1} \\
&\stackrel{\bullet}{\approx} \mathcal{C}'[\mathcal{C}_0[A_2]] \text{ by Lemma 6.2}
\end{aligned}$$

□

7 Related Work

Focardi and Gorrieri [15, 16] introduced nondeterministic noninterference (NNI) and variants (SNNI, BNNI, etc) as a generalization of noninterference for concurrent and therefore nondeterministically interacting processes. Besides NNI, they also considered several other notions of information flow security, including for synchronous communication (SNNI) and bisimulation-based variants of these conditions (BNNI, BSNNI and other conditions). They also introduce the notion of BNDC equivalence: a process P is BNDC if what a low level user sees of the system is not modified by composing that system with any high level process. Busi and Gorrieri [8] consider noninterference for Petri net semantics, a popular operational model for true concurrency semantics. They show that the notion of BNDC for Petri nets is completely characterized by causality (high inputs enabling low outputs) and conflict (between high and low inputs for a transition). Busi and Gorrieri demonstrate that not all causalities and conflicts between high and low give rise to interference (for example, where a transition with high input is already enabled by the initial marking). These examples are related to work in interleaving models where some synchronization (for mutual exclusion) is allowed between high and low processes that access shared variables [17, 24].

Crafa and Rossi [11], building on the BNDC notion of correctness, investigate the notion of “controlled information release” in a type π -calculus extended with an explicit declassification expression. They allow

a low action succeeding a high action, provided there is an intervening declassification operation. The declassification operation is only available to “high” (trusted) processes, and “low” processes are not able to view the declassification itself. This may require the addition of nondeterminism in some cases to mask the fact of declassification. Sewell and Vitek introduce the box- π process calculus to express wrappers encapsulating trusted/untrusted components intended for security policies enforcement [35]. They present a causal type system that statically captures legitimate flows between components, although it is not clear what the security guarantees of the system are.

A great deal of work on information flow control has been done in the concurrency community. Hennessy and Riely [20, 21] develop a security π -calculus for which they study noninterference properties with respect to *may* and *must* testing. Honda and Yoshida [23] design a sophisticated system with linear and affine types for π -calculus to investigate noninterference expressed in terms of bisimulation. Boudol and Castellani [7] present a simple imperative language extended with parallelism to explore noninterference in a probabilistic setting. Ryan and Schneider characterize the absence of information flow in *CSP* [31] based on the notion of process equivalence. Bossi, Piazza and Rossi [1] generalize an unwinding framework for the definition of a security property that entails a noninterference principle described in a simple concurrent language. Similar approach related to this line of work can also be found in [10]. Most of these works are focused on strong noninterference properties usually characterized by a partial equivalence relation in a typed process language.

To avoid termination leaks in concurrency, Smith and Vopano [36] investigated noninterference in a constrained multi-threaded language without a high guard in *while* loops. Later, Smith, and independently, Boudol and Castellani [7] relaxed this constraint but only allow such a statement to be followed by a high statement, if any. Sabelfeld [32] simulates such *while* loops as an effect of synchronization, in addition to the restriction above. High to low process synchronization is impossible due to the confinement of semaphores. In all these multi-threaded languages, variant conservative security type systems are constructed to rule out potential termination leak by restricting programs to certain behaviors. To enhance expressiveness, Sabelfeld and Mantel [33] introduce encryption and more communication primitives in their multi-threaded language. In security process calculi, Hennessy identifies a similar problem in designing multi-security levels in [20], where the strong noninterference property cannot be achieved because the contention on different security level channels reveals distinct timing behavior to observer. Focardi et al. [16, 17] propose a *program transformation* approach to mask such activity by including execution possibilities. Another trend in permitting secure communication such as synchronization and deadlock-freedom is to relieve the restrictions, such as in [29, 20, 21], where advanced type systems are designed to enforce security properties. The goal of secure nested transactions is to provide a relatively simple and familiar type system, with the dynamic semantics enforcing properties that would in the aforesaid approaches be enforced using type-checking. [40, 23, 22, 24]. The relationship between our work and these other works is demonstrated by for example the fact that our proof technique for noninterference follows the technique used by Kobayashi for a system with linear types [24], where the latter is in turn based on the approach of Pottier [29].

To monitor information flow of a concurrent program caused by synchronization for example, in [19] the author develops a method, automaton-based confidentiality monitoring, that combines dynamic and static analysis to control program execution in accordance with noninterference. Researchers in [28] explore a static analysis for π -calculus with which they investigate program privacy in flow-sensitive and context-sensitive configurations. The resulted approach can track the flow of information and model its action order, based on which to validate privacy property in communication including synchronization. For securing information release of a dynamic language, particularly in termination-sensitive context, Askarov and Sabelfeld [2] impose a restriction to disallow dynamic primitive such as *eval* in branches of high condition-

als, as a standard treatment in [36].

Bertino et al [5] consider noninterference for nested transactions. They are principally concerned with the issue of starvation of high transactions in multi-level databases. Rather than pre-empting a high transaction if it holds a lock that a low transaction requires, they introduce “signal locks” which are essentially callbacks that low transactions use to notify high transactions of updates on shared variables. This allows high transactions to decide what to do when they dynamically discover a race condition. The aforesaid paper extends the approach of signal locks from flat to nested transactions. If a signalled high transaction chooses not to abort, there is no guarantee of serializability between transactions at different levels. If they choose to abort, then there is no longer any guarantee of lack of starvation for high transactions. This work does not consider the issue of locks anti-inherited from high to low transactions, which is the motivation for retroactive abort. In the locking rules in Sect 4.2 in [5], a transaction is not able to obtain a lock “retained” by another transaction, unless the latter transaction is one of its ancestors. In terms of the example in Fig. 1, transaction T_2^L would be blocked from acquiring the lock implicitly held by transaction T_1^L .

Birgisson and Erlingsson in [6] present the semantics and implementation for transactional memory introspection (TMI) that supports the enforcement of security policies in a reflective software transactional memory. Intransitive noninterference [30, 37] has been applied to information flow control to require intervening processing before information is declassified. An example is adding a Password level to the standard lattice of Low and High, and allowing flows from High to Low through Password (requiring a dynamic password check), but not directly from High to Low. Cohen, Meyden and Zuck [9] develop an access control model based on intransitive noninterference [30, 37] to reason about allowable information flows due to access aborts in software transactional memory. All of these works essentially adapt the work of multilevel databases to software transactional memory. None of them consider nested transactions.

8 Conclusions

This work synthesizes two strands of research in computer security: multilevel secure databases, a field at least twenty years old, and language-based security, a more recent focus of security research. Secure nested transactions constitute a new approach to combining low and high processes in systems where we wish to prevent information flow leaks. Processes at different security levels are able to (implicitly) synchronize without leaking information. The use of nested transactions, a new consideration in this article, is crucial for allowing this mixing of security levels, so high and low transactions can safely collaborate. Because process aborts are now observable, nesting of transactions must replace the stacking of security contexts in the sequential case. Synchronization leaks are prevented by giving low transactions the power to abort high transactions that attempt to hoard resources. This notion of low transactions pre-empting high transactions is already well-known from multilevel databases [3]. *Retroactive abort* is a new approach introduced in this article, that extends the existing approaches to nested transactions.

Our semantics includes certain messages considered as “locks”. However these should really be viewed as a special kind of message, where the transactional semantics ensures that these messages are treated linearly. It is this linearity in the processing of these messages in the semantics that ensures noninterference, and so allows communication from low to high processes without leaking information via traffic analysis. Lock-based synchronization on the other hand is not a critical aspect of the semantic. Indeed it should be straightforward to extend the results to timestamp-based and version-based concurrency control, by adapting the log conditions on visibility of effects, without modifying the semantic reaction and reduction rules. The main challenge is in incorporating these semantics into nested transactions, and there is much work on the latter to be availed of.

The implementation of secure nested transactions does not pose significant challenges, beyond those posed by nested transactions themselves. Retroactive abort is only possible within the context of a single root transaction. Failure dependencies do not cross the boundaries of such transactions, therefore we do not have cascading aborts. Our semantics is deliberately designed to be low-level, in terms of a message-passing calculus and logs, both of which are standard in distributed systems technologies, so that the implementation is straightforward. We even avoid synchronous message-passing. The main challenge is in checking non-local log conditions. This is already part of the semantics of nested transactions, using two-phase commit to commit the root transaction, which may necessitate the abort of some tentative commits.

In related work, we have developed a “global” version of this semantics, where all logs are at “top-level.” This semantics is greatly simplified by avoiding the need for logical constraints on reaction rules in the semantics. Instead the logs are tested directly for preconditions. This simplification is possible because in the global system we are not concerned about reasoning compositionally about observational equivalence for the purpose of verifying noninterference. This is related to the technique of input-output automata used to reason about classical nested transactions [25]. Using this system, we are able to verify serializability and durability results. We are also able to relate that global system to the local system presented here. We hope to have the opportunity to report on these results at some point in the future.

We view secure nested transactions as a baseline for ensuring lack of information leaks. An obvious direction for future work is to consider intransitive policies. Another direction to consider is the synthesis of secure nested transactions with type-based approaches to secure synchronization [24]. The latter statically ensures lack of information leaks, while the former relies on dynamic mechanisms for preventing leaks. A hybrid system that synthesized these approaches would be a fruitful avenue for future work.

References

- [1] C. Piazza A. Bossi and S. Rossi. Compositional information flow security for concurrent programs. *Journal of Computer Security*, 15(3):373–416, 2007.
- [2] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 43–59, Washington, DC, USA, 2009. IEEE Computer Society.
- [3] V. Atluri, S. Jajodia, and B. George. *Multilevel Secure Transaction Processing*. Kluwer Academic Publishers, 1999.
- [4] Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, 2005.
- [5] Elisa Bertino, Barbara Catania, and Elena Ferrari. A nested transaction model for multilevel secure database management systems. *ACM Trans. Inf. Syst. Secur.*, 4:321–370, November 2001.
- [6] Arnar Birgisson and Úlfar Erlingsson. An implementation and semantics for transactional memory introspection in haskell. In *PLAS*, pages 87–99, 2009.
- [7] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.*, 281(1-2):109–130, 2002.
- [8] Nadia Busi and Roberto Gorrieri. Positive non-interference in elementary and trace nets. In *ICATPN, LNCS*, pages 1–16. Springer-Verlag, 2004.

- [9] Ariel Cohen, Ron van der Meyden, and Lenore D. Zuck. Access control and information flow in transactional memory. In *Formal Aspects in Security and Trust (FAST)*, 2008.
- [10] S. Crafa and S. Rossi. A theory of noninterference for the pi-calculus. In *Proc. of the Symposium on Trustworthy Global Computing (TGC'05)*, volume 3705 of *LNCS*, pages 2–18. Springer-Verlag, 2005.
- [11] S. Crafa and S. Rossi. Controlling information release in the pi-calculus. *Information and Computation*, 285(8):1235–1273, 2007.
- [12] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19, 1976.
- [13] D. E. Denning and P. J. Denning. Certifications of programs for secure information flow. *Communications of the ACM*, 20(7), 1977.
- [14] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Symposium on Operating Systems Principles (SOSP)*, 2005.
- [15] R. Focardi and R. Gorrieri. A taxonomy of security properties for process algebras. *Journal of Computer Security*, 3(1):5–34, 1995.
- [16] R. Focardi and R. Gorrieri. Classification of security properties (part i: Information flow). In *FOSAD*, number 2171 in *LNCS*, pages 331–396. Springer-Verlag, 2000.
- [17] R. Focardi and S. Rossi. Information flow security in dynamic contexts. In *Computer Security Foundations Workshop*, pages 307–319. IEEE Press, 2002.
- [18] R. Focardi, S. Rossi, and A. Sabelfeld. Bridging language-based and process calculi security. In *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, number 3441 in *Lecture Notes in Computer Science*, pages 299–315. Springer-Verlag, 2005.
- [19] Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Symposium (CSF)*, pages 218–232. IEEE Computer Society, 2007.
- [20] Matthew Hennessy. The security picalculus and non-interference. *Journal of Logic and Algebraic Programming*, 63:3–34, 2004.
- [21] Matthew Hennessy and James Riely. Information flow vs resource access in the asynchronous pi-calculus. *toplas*, 24(5):566–591, 2002.
- [22] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *ESOP*, volume 1782 of *LNCS*, pages 180–199. Springer-Verlag, 2000.
- [23] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *POPL*, pages 81–92. ACM, 2002.
- [24] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 2003.
- [25] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan-Kaufman, 1994.

- [26] E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.
- [27] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 228–241, 1999.
- [28] Hanne Riis Nielson and Flemming Nielson. A flow-sensitive analysis of privacy properties. In *CSF*, pages 249–264. IEEE Computer Society, 2007.
- [29] Francois Pottier. A simple view of type-secure information flow in the pi;-calculus. In *In Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 320–330, 2002.
- [30] J. M. Rushby. Noninterference, transitivity and channel-control security policies. Technical report, SRI, 1992.
- [31] P Y A Ryan and S A Schneider. Process algebra and non-interference. In *CSFW '99: Proceedings of the 12th IEEE workshop on Computer Security Foundations*, page 214, Washington, DC, USA, 1999. IEEE Computer Society.
- [32] A. Sabelfeld. *Semantic Models for the Security of Sequential and Concurrent Programs*. PhD thesis, Chalmers University of Technology and Gothenburg University, Gothenburg, Sweden, May 2001.
- [33] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *SAS*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, 2002.
- [34] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2002.
- [35] Peter Sewell and J. Vitek. Secure composition of untrusted code: Wrappers and causality types. In *Computer Security Foundations Workshop*, 2000.
- [36] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 19–21, 1998.
- [37] R. van der Meyden. What, indeed, is intransitive noninterference? In *European Symposium on Research in Computer Security (ESORICS)*. Springer-Verlag, 2007.
- [38] D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*. Springer-Verlag, 1997.
- [39] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.
- [40] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *In Proc. 16th IEEE Computer Security Foundations Workshop*, pages 29–43, 2003.
- [41] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. Securing distributed systems with information flow control. In *Network Systems Design and Implementation*, 2008.