Corrective Synchronization via Trace Warping

Anonymous

1. Introduction

Concurrency control is a hard problem. While some thread interleavings are admissible (in particular, if they involve disjoint memory accesses), there are certain interleaving scenarios that must be inibited to ensure serializability [**?**]. The goal is to automatically detect — with high precision and low overhead — the inadmissible interleavings, and ensure that that do not take effect.

Toward this end, there are currently two main synchronization paradigms:

- *Pessimistic synchronization*: In this approach, illegal interleaving scenarios are avoided conservatively by blocking the execution of one or more of the concurrent threads until the threat of incorrect execution has passed away. Locks, mutexes and semaphores are all examples of how to enforce mutual exclusion, or pessimistic synchronization.
- *Optimistic synchronization*: As an alternative to proactive, or pessimistic, synchronization, optimistic synchronization is essentially a reactive approach. The concurrency control system monitors execution, such that when an illegal interleaving scenario arises, it is detected as such and appropriate remediation steps are taken. A notable instance of this paradigm is transactional memory (TM) [**?**], where the system logs memory accesses by each of the threads, and is able to reverse the effects of a thread and abort/restart it.

Motivation The pessimistic approach is useful if critical sections are short, there is little available concurrency, and the involved memory locations are well known [**?**]. Optimistic synchronization is most effective when there is a high level of available concurrency. An example is graph algorithms, such as Boruvka, over graphs that are sparse and irregular [**?**].

Beyond these cases, however, there are many other situations of practical interest. As an illustrative example, we refer to the code fragment in Figure 1, extracted from the dyuproject project, where a shared `Map` object, (pointed-to by) _convertors, is manipulated by method getConvertor().

Assume that different threads invoking this method are all attempting to simultaneously obtain the same Convertor object, which has not yet been created. Doing so optimistically would lead to multiple rollbacks, and thus poor performance. Mutual exclusion, on the other hand, would block all threads but one until the operation completes, which is far from optimal if newConvertor() is an expensive operation.

```
public Convertor getConvertor(
      Class cls ,boolean create,boolean add) {
  Convertor convertor = _convertors.get( cls .getName());
  if (convertor==null && create) {
    convertor = newConvertor(cls,add);
    _convertors . putIfAbsent( cls .getName(), convertor ); }
   return convertor ; }
```

Figure 1: Method getConvertor() from class StandardConvertorCache in project dyuproject

Our Approach We propose a novel synchronization paradigm, which is conceptually different from both the pessimistic and the optimistic approaches. In our approach, dubbed *corrective synchronization*, the correctness of multi-threaded execution is enforced after the fact, similarly to optimistic synchronization, though without rollbacks. Instead, the system automatically compensates, if necessary, for the effects of inadmissible interleavings by rewriting the program state as a transaction completes. This is done while accounting for the behavior of concurrent transactions, so as to guarantee serializability.

To illustrate our approach, we revisit the running example. Assume the following execution history:

T_1	T_2
_convertors.get()/null	
	_convertors.get()/null
if(...)	
	if(...)
newConvertor()/o_1	
	newConvertor()/o_2
_convertors.putIfAbsent()/null	
	_convertors.putIfAbsent()/o_1
return o_1	
	return o_2

This history is clearly nonserializable, as in any serializable history T_1 and T_2 would return the same Convertor instance. Correcting this execution involves the application of two actions to the exit state of T_2. First, we point the local variable convertor to o_1, rather than o_2. Second, we fix the mapping under $\_convertors$ for key cls.getName() in the same way.

Note that the corrective actions above are of a general form, which is not limited to only two threads. For any number of threads, the corrected state would have one privileged thread deciding the return value (i.e., the value of convertor) for all threads, which would also be the value linked by the key under _convertors. Also note that the corrective actions are — relatively speaking — inexpensive, especially compared to the alternatives of either blocking or aborting/restarting all threads but one.

tbd tbd

Two important challenges that we address in this paper w.r.t. the corrective synchronization paradigm are (i) how to compute correct poststates; and (ii) given an incorrect poststate, how to decide which correct poststate to transition to. We govern our discussion of these challenges by a formal framework, based on the push/pull model for transactions [**?**], with rigorous soundness guarantees. We also provide a clear statement of the limitations of corrective synchronization.

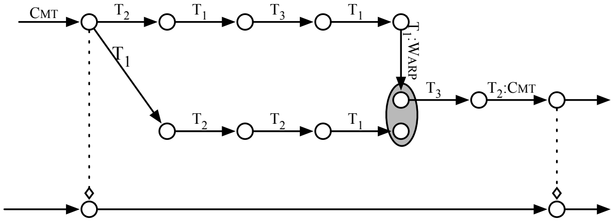
Beyond the formal details, this paper additionally addresses practical challenges, and in particular the question of how to implement corrective synchronization efficiently, such that it incurs low overhead. To this end, we present a solution based on static analysis to derive the correct poststates in relation to a given prestate. We have implemented a version of the analysis for shared Map data structures. Maps are used to represent the shared state of many Java programs [**?**], and so this first step toward a comprehensive static analysis for corrective synchronization is already of practical value.

say that the implementation does something very simple (assumes one transaction), yet illustrates something very powerful that is completely novel

Contributions This paper makes the following principal contributions:

1. Corrective synchronization: We present an alternative to both the pessimistic and the optimistic synchronization paradigms, dubbed *corrective synchronization*, whereby serializability is achieved neither via mutual exclusion nor via rollbacks, but through correction of the poststate according to a relational prestate/poststates specification.
2. Formal guarantees: We provide a formal description of corrective synchronization in terms of the push/pull model for transactions. This includes a correctness (or soundness) proof as well as a clear statement of limitations.
3. Static analysis: We have developed a static analysis to derive the prestate/poststates specification for programs that encode the shared state as one or more assoiciative mappings. We describe the analysis in full formal detail.
4. Implementation and evaluation: We have created a protoype implementation of corrective synchronization assuming the shared state is represented as associative mappings. We discuss techniques and optimizations to achieve low overhead. We present experimental evidence in favor of corrective synchronization, where our subjects are derived from real-world Java applications.

2. Overview



3. Technical Background

In this section we describe a generic language of transactions and define an idealized semantics for concurrent transactions called the atomic semantics, in which there are no interleaved effects on the shared state. The model preliminaries generalize those provided previously [**?**]. We also define a notion of *good* configurations and in the next section we will define how one can warp from a configuraiton that is not good to one that is.

Operations and States. We assume a set M of method calls or operations (*e.g.* `ht.put('a',5)`). State is represented in terms of logs of operation records. An operation record (or, simply, an "operation") $op = \langle m, \sigma_1, \sigma_2, id\rangle$ is a tuple consisting of the operation name m, a thread-local pre-stack σ_1 (method arguments), a thread-local post-stack σ_2 (method return values), and a unique identifier id. We assume a predicate $\mathsf{fresh}(id)$ that holds provided that id is globally unique (details omitted for lack of space). In the atomic semantics defined below, the shared state $\ell : \mathsf{list}\ op$ is an ordered list of operations. We use notations such as $\ell_1 \cdot \ell_2$ and $\ell \cdot op$ to mean append and appending a singleton, resp.

We require a prefix-closed predicate on operation lists $\mathsf{allowed}\ \ell$ that indicates whether an operation log ℓ corresponds to a state. For convenience we will also write ℓ allows $\langle m, \sigma_1, \sigma_2, id\rangle$ which simply means $\mathsf{allowed}\ \ell \cdot \langle m, \sigma_1, \sigma_2, id\rangle$. For example, if we have a simple TM based on memory read/write operations we expect $\mathsf{allowed}\ \ell \cdot \langle \mathtt{a\ :=\ x}, [x \mapsto 5], [x \mapsto 5, a \mapsto 5], id\rangle$, but $\neg\mathsf{allowed}\ \ell \cdot \langle \mathtt{a\ :=\ x}, [x \mapsto 5], [x \mapsto 5, a \mapsto 3], id\rangle$ or more elaborate specifications that involve multiple tasks. Ultimately, we expect the $\mathsf{allowed}$ predicate to be induced by the implementation's operations on the state, $[\![op]\!] : \mathcal{P}(\mathsf{State} \times \mathsf{State})$, and initial states I.

We define a precongruence over operation logs $\ell_1 \preccurlyeq \ell_2$ coinductively, by requiring that all $\mathsf{allowed}$ extensions of the log ℓ_1, are also $\mathsf{allowed}$ extension to the log ℓ_2. We use a coinductive definition so that the precongruence can be defined up to all infinite suffixes.

$$\frac{\mathsf{allowed}\ \ell_1 \Rightarrow \mathsf{allowed}\ \ell_2 \quad \forall op.\ (\ell_1 \cdot op) \preccurlyeq (\ell_2 \cdot op)}{\ell_1 \preccurlyeq \ell_2}$$

We use a double-line here to indicate greatest fixpoint. Informally, the above definition says that there is no sequence of observations we can make of ℓ_2, that we can't also make of ℓ_1. This is more general than just considering the set of states reached from executing the first log is included in the second: unobservable state differences are also permitted.

Language. Threads execute code c from some programming language that includes thread forking, transactions $\mathtt{tx}\ c$, method names such as m, and a `skip` statement. As done elsewhere [**?**], we abstract away the programming language with a few semantic functions: update this with pldi camera ready

$c \lightning\lightning (m, c')$**:** Within a transaction, code c can be reduced to the pair (m, c'). That is, m is a next reachable method call in the reduction of c, with remaining code c'.

$c \lightning (t, c')$**:** Outside of a transaction, code c can be reduced to the pair (t, c'). Here c' is the remaining code, and t is either a local state update, or a transaction or a thread fork.

$\mathbf{fin}(c)$**:** This predicate is true provided that there is a reduction of c to `skip` that does not encounter a method call.

These functions allow us to obtain a simple semantics, despite an expressive input language, by introducing functions to resolve nondeterminism between method operation names and at the end of a transaction. We assume that code is well-formed in that a single operation name m is always contained within a transaction.

Atomic Transition Systems. We next define a simple atomic semantics $\xrightarrow{a}$, in which transactions are executed instantly, without interruption from concurrent threads.

The $\xrightarrow{a}$ rules AFIN, AFORK, ALOCAL showin in Figure 2(b) are similar to their counterparts in $\xrightarrow{u}$. However, the ATXN rule says that if thread executing code c_1 can reduce to a transaction

(a) **Atomic Machine Big Step Transaction Rules** $\Downarrow$

$$\frac{\mathsf{fin}(c)}{(c,\sigma),\ell \Downarrow c,\ell}\ \textsc{BSFin} \qquad \frac{c\lightning\lightning(m,c_2) \quad \ell_1\ \mathsf{allows}\ \langle m,\sigma,\sigma'\rangle \quad (c_2,\sigma'),\ell\cdot[\langle m,\sigma,\sigma'\rangle] \Downarrow \sigma'',\ell_2}{(c,\sigma),\ell_1 \Downarrow \sigma'',\ell_2}\ \textsc{BSStep}$$

(b) **Atomic Machine Rules** $\xrightarrow{a}$

$$\frac{\mathsf{fin}(c)}{\mathbf{A}_1\cdot(c,\sigma)\cdot\mathbf{A}_2,G \xrightarrow{a} \mathbf{A}_1\cdot\mathbf{A}_2,G}\ \textsc{AFin}$$

$$\frac{c_1\lightning(\texttt{fork}\ \ c,c_2)}{\mathbf{A}_1\cdot(c,\sigma)\cdot\mathbf{A}_2,G \xrightarrow{a} \mathbf{A}_1\cdot(c_2,\sigma)\cdot(c',\sigma)\cdot\mathbf{A}_2,G}\ \textsc{AFork}$$

$$\frac{c\lightning(\texttt{local}\ \ R,c') \quad R\,\sigma\,\sigma'}{\mathbf{A}_1\cdot(c,\sigma)\cdot\mathbf{A}_2,G \xrightarrow{a} \mathbf{A}_1\cdot(c',\sigma')\cdot\mathbf{A}_2,G}\ \textsc{ALocal}$$

$$\frac{c_1\lightning(\texttt{tx}\ c',c_2) \quad (c',\sigma),G \Downarrow \sigma',G'}{\mathbf{A}_1\cdot(c_1,\sigma)\cdot\mathbf{A}_2,G \xrightarrow{a} \mathbf{A}_1\cdot(c_2,\sigma')\cdot\mathbf{A}_2,G'}\ \textsc{ATxn}$$

Figure 2: Atomic semantics of concurrent threads.

`tx` c', then the transaction c' is executed atomically by the big step rules $\Downarrow$ described next.

Figure 2(a) illustrates the big step semantics $\Downarrow$, which uses $\lightning\lightning$ and $\mathsf{fin}()$ (rules BSStep and BSFin, respectively). These rules scan through the nondeterminism in `tx` c to find a next operation name m or a path to `skip` denoting the end of the transaction. BSStep can be taken provided that the operation $\langle m,\sigma,\sigma'\rangle$ is permitted and that (c_2,σ') can be entirely reduced to (σ'',ℓ_2).

4. Warping

The key idea of this paper is that when transactions run awry into an inconsistent state, rather than aborting them and starting again from the beginning, it may be possible to *correct* the shared/local state directly by directly modifying the state. Thus, the program continues as if it had not gone down the bad path to the inconsistent state.

We now formalize a simple version where, from an inconsistent state, the system warps to a state that was reachable in a serial interleaving. We later show this can be easily generalized to allow warping to more subtle states (such as those described by the Push/Pull model []). These *super*-serial (yet still serializable) allow for more possible destinations, making it more likely that a destination will be available when an inconsistent state is reached.

Unconstrained transition system. We begin with a generic transition system $\xrightarrow{u}$ in which threads may interleave their effects however they please:

$$\frac{\mathsf{fin}(c)}{\mathbf{A}_1\cdot(c,\sigma)\cdot\mathbf{A}_2,G \xrightarrow{u} \mathbf{A}_1\cdot\mathbf{A}_2,G}\ \textsc{UFin}$$

$$\frac{c_1\lightning(\texttt{fork}\ \ c,c_2)}{\mathbf{A}_1\cdot(c,\sigma)\cdot\mathbf{A}_2,G \xrightarrow{u} \mathbf{A}_1\cdot(c_2,\sigma)\cdot(c',\sigma)\cdot\mathbf{A}_2,G}\ \textsc{UFork}$$

$$\frac{c\lightning(\texttt{local}\ \ R,c') \quad R\,\sigma\,\sigma'}{\mathbf{A}_1\cdot(c,\sigma)\cdot\mathbf{A}_2,G \xrightarrow{u} \mathbf{A}_1\cdot(c',\sigma')\cdot\mathbf{A}_2,G}\ \textsc{ULocal}$$

$$\frac{c_1\lightning\lightning(m,c_2) \quad G\ \mathsf{allows}\ \langle m,\sigma,\sigma'\rangle}{\mathbf{A}_1\cdot(\texttt{tx}\ c_1,\sigma)\cdot\mathbf{A}_2,G \xrightarrow{u} \mathbf{A}_1\cdot(\texttt{tx}\ c_2,\sigma')\cdot\mathbf{A}_2,G\cdot\langle m,\sigma,\sigma'\rangle}\ \textsc{UTxn}$$

```
s ::= m.put(k, v)
    | v = m.get(k)
    | m.remove(k)
    | v = m.putIfAbsent(k, v)
    | v = new Value()
    | v = null
    | if(b) s₁; else s₂
    | while(b) s₁;
    | s₁; s₂

b ::= x == NULL | m.containsKey(k) | !b₁
```

Figure 3: Statements and conditions

The semantics is a relation $\xrightarrow{u}$ over pairs consisting of a list of concurrent threads $\mathbf{A}$ and a shared state ℓ. A single thread $(c,\sigma) \in \mathbf{A}$ is a code c and local state σ.

The unconstrained machine can take a UFin step when there is a thread (c,σ) that can complete, *i.e.* $\mathsf{fin}(c)$. The UFork rule allows a new thread (c',σ) to be forked from thread (c,σ). The ULocal rule involves manipulating the thread-local state σ to σ'. Finally, the UTxn rule allows a thread executing transaction code `tx` c_1 to take a single step to c_2, applying the effects of m directly to the shared log G.

Corrective Warping and Committing. A simple version of warping (which we have implemented in Section **??**) builds on an unconstrained transition system $\xrightarrow{u}$ by adding a special WarpCmt rule. This rule attempts to perform a warp— replacing the current state with state that would have been reached in an atomic interleaving and then committing—or else, aborts. This is already more expressive than all existing notions of transactions which, in the face of an inconsistent state perform a (potentially partial) abort.

add the fact that each thread remembers the $\mathbf{T},G$ where they began

$$\frac{\mathsf{fin}(c_1) \quad \mathrm{Warp}(\mathbf{A}_1\cdot(c_2,\sigma)\cdot\mathbf{A}_2,G,\mathbf{A}_1'\cdot(c_2',\sigma')\cdot\mathbf{A}_2',G')}{\mathbf{A}_1\cdot((\texttt{tx}\ c_1,c_2),\sigma)\cdot\mathbf{A}_2,G \xrightarrow{u} \mathbf{A}_1'\cdot(c_2',\sigma')\cdot\mathbf{A}_2',G'}\ \textsc{WarpCmt}$$

how to jump to an atomic state? you haven’t committed yet.
reference state issue.
conditions on warping - consistency.

5. Language

As a proof of concept and preliminary practical study, we instantiate the theoretical framework we formalized in the Section 4 on the language in Figure 3, developing a static analysis to compute warping **targets**, and a dynamic system to warp.

The language is focused on a representative set of operations of the Java `Map` interface. In Figure 3, we represent by `m` the map shared among all the transactions, and `k` the shared key. The values inserted or read from the map might be a parameter of the transaction, or created through a `new` statement. Following the semantics of the Java library, our language supports (i) `v = m.get(k)` that returns the value `v` related with key `k`, or `null` if `k` is not in the map, (ii) `m.remove(k)` removes `k` from the map, (iii) `v = m.putIfAbsent(k, v)` relates `k` to `v` in `m` if `k` is already in `m` and returns the previous value it was related to, (iv) `v = new Value(...)` creates a new value, and (v) `v = null` assigns `null` to variable `v`. In addition, our language support standard `if` and `while` statements, as well as concatenation of statements.

As Boolean conditions, the language supports checking if a variable is `null`, and if the map contains a key.

```
Value removeAttribute(Key k) {
  Value result = null;
  if(map.containsKey(k)) {
    result = m.get(k);
   m.remove(k);
  }
  return result ;
}

boolean removeAttribute(Key k, Value v) {
  Value oldvalue = m.get(k);
  m.put(k, v);
  return oldvalue != null;
}
```

Figure 4: The running example inspired by class `ApplicationContext` of Apache Tomcat

5.1 Running Example

Figure 4 illustrates our running example. This code is inspired by **XXX**. The first type of transaction (`transaction1`) removes the value associated with the given key `k`, and returns it. Instead, the second type of transaction relates `k` with a given value `v`, and returns `true` if the key was already in the map. During the formalization of the static analysis and the warping system, we will refer to this running example where each transaction is instantiated multiple times, and all transactions conflict on the same key `k`.

6. Static Computation of Warp Destinations

6.1 Abstract Domain

Let Var and HeapNode be the set of variables and abstract heap nodes, respectively. We suppose that a special `null` value is part of HeapNode. Both keys and values are abstracted as heap nodes. As usual with heap abstractions, each heap node might represent one or many concrete nodes. Therefore, we suppose that a function $\mathsf{summary} : \mathsf{HeapNode} \rightarrow \{\mathsf{true}, \mathsf{false}\}$ is provided; $\mathsf{summary}(h)$ returns `true` if and only if h represents many concrete nodes (that is, it is a summary node). We define by $\mathsf{Env} : \mathsf{Var} \rightarrow \wp(\mathsf{HeapNode})$ the set of (abstract) environments relating each variable to the set of heap nodes it might point to. A map is represented as a function $\mathsf{Map} : \mathsf{HeapNode} \rightarrow \wp(\mathsf{HeapNode})$, connecting each key to the set of possible values it might be related to in the map. The value `null` represents that the key might not be in the map. For instance, $[n_1 \mapsto \{\mathtt{null}, n_2\}]$ represents that the key n_1 might not be in the map, or it is in the map, and it is related to value n_2. An abstract state is a pair made by an abstract environment and an abstract map. We augment this set with a special bottom value $\bot$ to will be used to represent that a statement is unreachable. Formally, $\Sigma = (\mathsf{Env} \times \mathsf{Map}) \cup \{\bot\}$.

The lattice structure is obtained by the point-wise application of set operators to elements in the codomain of abstract environments and functions. Therefore, the abstract lattice is defined as $\langle \Sigma, \dot{\subseteq}, \dot{\cup} \rangle$, where $\dot{\subseteq}$ and$\dot{\cup}$ represents the point-wise application of set operators $\subseteq$ and $\cup$, respectively.

Running example. Consider the first method in Figure 4. For instance, the abstract state $([\mathtt{k} \mapsto \{n_1\}], [n_1 \mapsto \{\mathtt{null}\}])$ represents that the key `key` is not in the map, while $([\mathtt{k} \mapsto \{n_1\}], [n_1 \mapsto \{n_2\}])$ represents that it is in the map, and it is related to a value. Instead, for the second method, $([\mathtt{k} \mapsto \{n_1\}, \mathtt{v} \mapsto n_2], [n_1 \mapsto \{n_2\}])$ represents that `k` is in the map, and it is related to the value pointed by `v`

6.2 Abstract Semantics

Figure 5 formalizes the abstract semantics of statements and Boolean conditions, that, given an abstract state (as defined in Section 6.1) and a statement or Boolean condition of the language introduced in Section 5, returns the abstract state resulting from the evaluation of the given statement on the given abstract state. We focus the formalization on abstract states in $\mathsf{Env} \times \mathsf{Map}$, since in case of $\bot$ the abstract semantics always returns $\bot$ itself.

(`put`) relates `k` to `v` in the map. In particular, if `k` points to a unique concrete node, it performs a so-called strong update, overwriting previous values related with `k`. Otherwise, it performs a weak update by adding to the previous values the new ones. (`get`) relates the assigned variable `v` to all the heap nodes of values that might be related with `k` in the map. Note that if `k` is not in the map, then the abstract map m relates it to a `null` node, and therefore this value is propagated to `v` then calling `get`, representing the concrete semantics of this statement. Similarly to (`put`), (`rmv`) removes `k` from the map (by relating it to the singleton $\{\mathtt{null}\}$) iff `k` points to a unique concrete node. Otherwise, it adds the heap node `null` to the heap nodes related to all the values pointed by `k`. (`pIA`) updates the map like (`put`) but only if the updated key node might have been absent, that is, when $\mathtt{null} \in m(n)$. `new` creates a new heap node through $\mathsf{fresh}(t)$ (where t is the identifier of the transaction performing the creation), and assigns it to `v`. The number of nodes is kept bounded by parameterizing the analysis with an upper bound i such that (i) the first i nodes created by a transaction are all concrete nodes, and (ii) all the other nodes are represented by a summary node. Instead, (`null`) relates the given variable to the singleton $\{\mathtt{null}\}$. Rules (`if`), (`while`), and (`cnc`) define the standard abstract semantics of `if`, `while`, and concatenation statements. The abstract semantics on Boolean conditions produces $\bot$ statements if the given Boolean condition cannot hold on the given abstract semantics. Therefore, (`null`) returns $\bot$ if the given variable `x` cannot be `null`, or a state relating `x` to the singleton $\{\mathtt{null}\}$ otherwise. Vice-versa, (!`null`) returns $\bot$ if `x` can be only null, or a state relating `x` to all its previous values except `null` otherwise. Similarly, (`cntK`) returns $\bot$ if the given key `k` is surely not in the map, it refines the possible values of `k` if it is represented by a concrete node, or it simply returns the entry state otherwise. Vice-versa, (!`cntK`) returns $\bot$ if `k` is surely in the map.

Running example. Consider again the first method in Figure 4. When we start from the abstract state $([\mathtt{k} \mapsto \{n_1\}], [n_1 \mapsto \{\mathtt{null}\}])$ (representing that `k` is not in the map), we obtain the abstract state $\sigma = ([\mathtt{k} \mapsto \{n_1\}, \mathtt{result} \mapsto \{null\}], [n_1 \mapsto \{\mathtt{null}\}])$ after the first statement by rule (`null`). During the following computation of rule (`if`), we consider:

1. when the Boolean condition `map.containsKey(k)` holds. When applying rule (`cntK`) on σ we obtain $\bot$ since the node pointed by `k` is related to the singleton $\{\mathtt{null}\}$ in the map, representing that the map does not contain the key `k`; and
2. when !`map.containsKey(k)` holds. Rule (!`cntK`) applied to σ returns σ itself, since `k` is in relation only with `null` in the map.

(`if`) returns the upper bound of the two resulting states, that is $\bot \dot{\cup} \sigma = \sigma$, and the value pointed by `result` (that is, `null`) is returned. Therefore, our analysis computes on this example that, when the key is not in the map in the entry state, the method returns `null` and does not modify the map.

6.3 Over-approximating Serialized Executions

We now formalize how we compute an approximation of serialized executions relying on the abstract semantics defined in Section 6.2. Our system receives as input two types of transactions (e.g., the two

$$\mathbb{S}\llbracket \texttt{m.put(k,v)}, (e,m)\rrbracket = \begin{cases} (e, m[n \mapsto e(\texttt{v})]) & \text{if } e(\texttt{k}) = \{n\} \wedge \neg\mathsf{summary}(n) \\ (e, m[n \mapsto m(n) \cup e(\texttt{v}) : n \in e(\texttt{k})]) & \text{otherwise} \end{cases} \tag{put}$$

$$\mathbb{S}\llbracket \texttt{v = m.get(k)}, (e,m)\rrbracket = (e[\texttt{v} \mapsto \textstyle\bigcup_{n\in e(k)} m(n)], m) \tag{get}$$

$$\mathbb{S}\llbracket \texttt{m.remove(k)}, (e,m)\rrbracket = \begin{cases} (e, m[n \mapsto \{\texttt{null}\}]) & \text{if } e(\texttt{k}) = \{n\} \wedge \neg\mathsf{summary}(n) \\ (e, m[n \mapsto m(n) \cup \{\texttt{null}\} : n \in e(\texttt{k})]) & \text{otherwise} \end{cases} \tag{rmv}$$

$$\begin{aligned} &\mathbb{S}\llbracket \texttt{v = m.putIfAbsent(k,v)}, (e,m)\rrbracket = (\pi_1(\mathbb{S}\llbracket \texttt{v = m.get(k)}, (e,m)\rrbracket), m') : \\ &\qquad m' = \begin{cases} (e, m[n \mapsto e(\texttt{v})]) & \text{if } e(\texttt{k}) = \{\texttt{n}\} \wedge m(n) = \{\texttt{null}\} \\ (e, m[n \mapsto m(n) \cup e(\texttt{v}) : n \in e(\texttt{k}) \wedge \texttt{null} \in m(n)]) & \text{otherwise} \end{cases} \end{aligned} \tag{pIA}$$

$$\mathbb{S}\llbracket \texttt{v = new Value()}, (e,m)\rrbracket = (e[v \mapsto \mathsf{fresh}(\texttt{t})], m) \tag{new}$$

$$\mathbb{S}\llbracket \texttt{v = new Value()}, (e,m)\rrbracket = (e[v \mapsto \{\texttt{null}\}], m) \tag{null}$$

$$\mathbb{S}\llbracket \texttt{if(b) s}_1\texttt{; else s}_2, (e,m)\rrbracket = \mathbb{S}\llbracket \texttt{s}_1, \mathbb{B}\llbracket \texttt{b}, (e,m)\rrbracket\rrbracket \dot\cup \mathbb{S}\llbracket \texttt{s}_2, \mathbb{B}\llbracket \texttt{!b}, (e,m)\rrbracket\rrbracket \tag{if}$$

$$\mathbb{S}\llbracket \texttt{while(b) s}_1\texttt{;}, (e,m)\rrbracket = \mathbb{B}\llbracket \texttt{!b}, (e_1,m_1)\rrbracket : (e_1,m_1) = \mathit{lfp}_{\bot}^{\dot\sqsubseteq} \lambda(e',m').(e,m) \dot\cup \mathbb{S}\llbracket \texttt{s}_1, \mathbb{B}\llbracket \texttt{b}, (e',m')\rrbracket\rrbracket \tag{while}$$

$$\mathbb{S}\llbracket \texttt{s}_1\texttt{; s}_2, (e,m)\rrbracket = \mathbb{S}\llbracket \texttt{s}_2, \mathbb{S}\llbracket \texttt{s}_1, (e,m)\rrbracket\rrbracket \tag{cnc}$$

$$\mathbb{B}\llbracket \texttt{x == null}, (e,m)\rrbracket = \begin{cases} (e[\texttt{x} \mapsto \{\texttt{null}\}], m) & \text{if } \texttt{null} \in e(\texttt{x}) \\ \bot & \text{otherwise} \end{cases} \tag{null}$$

$$\mathbb{B}\llbracket \texttt{!x == null}, (e,m)\rrbracket = \begin{cases} (e[\texttt{x} \mapsto e(\texttt{x}) \setminus \{\texttt{null}\}], m) & \text{if } \exists n \in \mathsf{HeapNode} : n \neq \texttt{null} \wedge n \in e(\texttt{x}) \\ \bot & \text{otherwise} \end{cases} \tag{!null}$$

$$\mathbb{B}\llbracket \texttt{m.containsKey(k)}, (e,m)\rrbracket = \begin{cases} \bot & \text{if } \forall n \in e(\texttt{k}) : m(n) = \{\texttt{null}\} \\ (e, m[n \mapsto m(n) \setminus \{\texttt{null}\}]) & \text{if } e(\texttt{k}) = \{n\} \wedge \mathsf{sum}\not\mathsf{m}\mathsf{ary}\ (n) \wedge m(n) \neq \{\texttt{null}\} \\ (e,m) & \text{otherwise} \end{cases} \tag{cntK}$$

$$\mathbb{B}\llbracket \texttt{!m.containsKey(k)}, (e,m)\rrbracket = \begin{cases} \bot & \text{if } \forall n \in e(\texttt{k}) : \texttt{null} \notin m(n) \\ (e, m[n \mapsto \{\texttt{null}\}) & \text{if } e(\texttt{k}) = \{n\} \wedge \mathsf{sum}\not\mathsf{m}\mathsf{ary}\ (n) \wedge \texttt{null} \in m(n) \\ (e,m) & \text{otherwise} \end{cases} \tag{!cntK}$$

Figure 5: Formal definition of the abstract semantics

methods in Figure 4), and returns a mapping from possible abstract entry states, to sets of possible exit states.

In particular, given two transactions `t1` and `t2`, we build up a control flow graph that represents possible serialized executions of many instances of the two transactions. In order to build up this serialized control flow graph, the local variables of transactions are rename. For instance, we can build up a serialized CFG where an instance of transaction `t1` (called $\texttt{t1}_1$) is followed by a loop representing a (possible unbounded) sequence of executions of other instances of the same transaction represented by a loop of a *summary* instance of `t1` (called $\texttt{t1}_\texttt{n}$), and this is followed by a similar structure for transaction `t2` (generating transactions $\texttt{t2}_1$ and $\texttt{t2}_\texttt{n}$).

When applied to these serialized control flow graphs, the static analysis engine produces exit states that are possible results of serialized executions. Given two transactions, we represent by $\mathsf{serializedCFGs}(\texttt{t1}, \texttt{t2}) = \mathsf{T}$ the function that returns a set of serialized CFGs.

6.4 Extracting Possible Warping States

As we discussed in **XXX**, we need to compute warping that, given an entry state representing an observational equivalence class, is in an equivalence class that is reachable through a serialized execution. However, an abstract state in Σ might represent concrete states that are in different equivalence classes. For instance $([\texttt{k} \mapsto \{n_1\}], [n_1 \mapsto \{\texttt{null}, n_2\}])$ represents both that `k` is (if n_1 is related to n_2 in the abstract map) or is not (when n_1 is related to `null`). This abstract state therefore might concretize to states belonging to different equivalence classes, and it cannot used to define a warping destination.

Therefore, we define a predicate $\mathsf{eqClass} : \Sigma \to \{\mathsf{true}, \mathsf{false}\}$ that, given an abstract state, holds iff it represents concrete states all in the same equivalence class. Formally,

$$\begin{gathered} \mathsf{eqClass}(e,m) \\ \Updownarrow \\ \forall \texttt{x} \in \mathit{dom}(e) : |e(\texttt{x})| = 1 \wedge e(\texttt{x}) = \{n_1\} \wedge \neg\mathsf{summary}(n_1) \\ \forall n \in \mathit{dom}(m) : |m(n)| = 1 \wedge m(n) = \{n_2\} \wedge \neg\mathsf{summary}(n_2) \end{gathered}$$

Prove that if $\mathsf{eqClass}(e,m)$ then all the concretized states from (e,m) are in the same equivalence class

Given two transactions `t1` and `t2`, we build up a set of possible entry states S such that $\forall (e,m) \in S : \mathsf{eqClass}(e,m)$. We then compute the exit states for all the possible serialized CFGs and entry states, and we filter our only the ones that represents states in the same equivalence class. The results are represented as a function that relates each entry state to a set of possible exit states. This is formalized by the following $\mathsf{warpDest}$ function.

$$\begin{aligned}&\mathsf{warpDest}(\mathtt{t1},\mathtt{t2},\mathsf{S}) = \{(e',m') :\\ &\quad \exists (e,m) \in \mathsf{S}, \exists \mathtt{s} \in \mathsf{serializedCFGs}(\mathtt{t1},\mathtt{t2}) :\\ &\qquad (e',m') \in \mathbb{S}[\![\mathtt{s},(e,m)]\!] \wedge \mathsf{eqClass}(e',m')\}\end{aligned}$$

Not yet explained how we compute the possible entry states

6.5 Concrete state

Let $\Sigma = \gamma(\mathbf{T} \times G)$ be the concrete states, denoted $\sigma = (\bar{t}, g)$.

Discover a $\mathcal{P}ietro : \hat{\Sigma} \to \hat{\Sigma} \to \wp(\hat{\Sigma})$, representing the current abstract state, the reference abstract state, and a set of possible destination abstract states.

7. Dynamic Warping

Given $\mathcal{P}ietro$, the runtime system implements a function denoted $\mathcal{P}eng : \sigma \to \hat{\sigma} \to \hat{\sigma} \to \sigma$.

Runtime tracks the current concrete state σ, current *abstract state* $\hat{\sigma}$ and the last *abstract reference state* $\hat{\sigma}_0$. Thus, we denote the runtime configuration as

$$c = \langle\!\langle \sigma, \hat{\sigma}, \hat{\sigma}_0 \rangle\!\rangle \quad \text{or, expanding } \langle\!\langle (\bar{t}, g), (\mathbf{T}, G), (\mathbf{T}_0, G_0) \rangle\!\rangle$$

That is, threads are in state $\bar{t}$, shared state g, tracked abstract state $(\mathbf{T}, G)$ and tracked abstract reference state $(\mathbf{T}_0, G_0)$.

There are then the following rules for steps in the runtime system:

$$\frac{\cdots}{\langle\!\langle \sigma, \hat{\sigma}, \hat{\sigma}_0 \rangle\!\rangle \hookrightarrow^* \langle\!\langle \sigma', \hat{\sigma}', \hat{\sigma}_0 \rangle\!\rangle}\ \textbf{Diverge}$$

$$\frac{\hat{\sigma}' \in \mathcal{P}ietro(\hat{\sigma}_0, \hat{\sigma}) \quad \sigma' = \mathcal{P}eng(\sigma, \hat{\sigma}, \hat{\sigma}')}{\langle\!\langle \sigma, \hat{\sigma}, \hat{\sigma}_0 \rangle\!\rangle \hookrightarrow \langle\!\langle \sigma', \hat{\sigma}', \hat{\sigma}_0 \rangle\!\rangle}\ \textbf{Warp}$$

$$\frac{\cdots}{\langle\!\langle \sigma, \hat{\sigma}, \hat{\sigma}_0 \rangle\!\rangle \hookrightarrow \langle\!\langle \sigma', \hat{\sigma}', \hat{\sigma}' \rangle\!\rangle}\ \textbf{Commit}$$

$$\frac{fix g \in \gamma(G) \quad \mathbf{T}, G \xrightarrow{PP} \mathbf{T}', G' \quad g' \in \gamma(G')}{\langle\!\langle \sigma, \hat{\sigma}, \hat{\sigma}_0 \rangle\!\rangle \hookrightarrow \langle\!\langle \sigma', \hat{\sigma}', \hat{\sigma}_0 \rangle\!\rangle}\ \textbf{Step}$$

$\mathcal{P}ietro$ ensures that $\hat{\sigma}'$ is reachable from $\hat{\sigma}_0$.

$\mathcal{P}eng$ ensures that $\sigma \in \gamma\hat{\sigma}$ and that you awlays warp before you commit (or you always eventually warp)

8. Experimental results

8.1 Implementation

Execution Summary The static analysis phase computes a set of execution summaries, each representing a legal execution, which are used as the input of the dynamic analysis phase. Each execution summary describes (i) the return value of each transaction instance and (ii) the final map state

In our experiments, we have two threads running two types of transactions (TX_1 and TX_2) respectively. One exemplary execution summary is, $[key \to v_1^1, r_1^1 : v_1^1, r_1^n : v_1^1, r_2^1 : v_1^1, r_2^n : v_1^1]$, where r_1^1 is the symbolic form of the return value of TX_1^1 ($1st$ instance of TX_1), v_1^1 is the symbolic form of the value put by the transaction instance, and $r_1^1 : v_1^1$ describes the return value of the transaction instance. The readers may notice r_1^n. This symbol represents the return value of the instances that are not explicitly specified. Here n is determined by the capability of the static analysis.

In addition, $key \to v_1^1$ tells what the key is associated with in the final map state.

The execution summary is not limited to the above basic case. In general, it is initial-state sensitive, schedule-oblivious and site-sensitive. First, the initial state about whether the key is mapped to some value affects the computation of the execution summaries. Therefore, we prefix each execution summary with a specific initial state, e.g., the state with initial key $[key^{init} \to v^{init}]$ or the state without initial key $[key^{INIT} \to VOID]$. Second, our execution summary is designed to be oblivious of the schedules. The schedule-obliviousness frees us from tracking the schedules at runtime and avoids the high tracking overhead. Third, the execution summary is site-sensitive. A transaction TX_1^1 may put a value dynamically created at site A to the map. We symbolically represent the value using the site and the occurrence of the site (inside the transaction instance), e.g., $A1$ in TX_1^1. The extent to which we can distinguish the occurrences is determined by the static analysis, e.g., how many iterations can be unrolled.

Runtime System The runtime works as follows. We first use a counter to track the transaction instance and assemble the symbolic return accordingly, e.g., r_1^1. Then we search for the symbolic value in the execution summaries, e.g., v_1^1. Last, using the symbolic value as a key, we look up the cache, which is maintained to associate each symbolic value with a runtime value, for the concrete runtime value.

The second step, i.e., searching for symbolic value in the summaries, is challenging. The searching is demanded on the fly at the return of each transaction instance. However, the returns should be consistent with each other such that the warped execution represents a realistic execution. In the other words, the return values should be searched for in the same execution summary. To achieve this, we implement an on-the-fly pruning algorithm. At the first return, it randomly picks the symbolic value. After the value is picked, the execution summaries with different value for the return will be pruned, leading to a smaller solution space. The algorithm is iteratively applied. In addition, to achieve initial-state sensitivity, we also reduce the solution space based on the initial states.

Another tricky issue is, at some return, the symbolic value found in the execution summaries may have not been associated with any concrete runtime value yet. For example, at the return of TX_2^1, we find the returned value as v_1^1 but TX_1^1 has not been executed, then we cannot find the runtime value associated with v_1^1 in the cache. This case may happen as the execution summary is computed for certain schedules, which differ from the current schedule. In this case, we apply the notify/wait primitives to synchronize the cache lookup and cache maintenance. Even more complex, the returned value may never be put into the cache if the value creation site is disabled in the current schedule (e.g., its guarding branch condition is false). We simply remove all the guarding branches for the creation sites. This simple strategy is sound as we treat the transaction as a blackbox and we only care about its return values. It is unnecessary to preserve the internal program semantics.

Evaluation

9. Related work

OMER

10. Conclusion

OMER

tree.jpg

Figure 6: An example

A. First Sketch

We start with a description of all possible interleaved executions of some bounded number of threads, as shown in Figure 6. Some of the leaf nodes denote admissible final states for the interleaved execution. The rest are inadmissible final states.

A generic concurrent system may produce both admissible and inadmissible executions. Two main approaches have been adopted to produce only admissible executions:

- apply synchronization to restrict the available parallelism and drive the system only into the admissible executions, and
- rollback the execution when the system falls into an inadmissible executions, and re-execute the program (sometimes driving the system to avoid to fall back to the inadmissible case already explored).

We would like to optimize two properties:

- Utilization of available parallelism, which is counted as the number of admissible paths that synchronization permits
- Overhead, which is counted as the number of operation/state reversals that synchronization may force to (re)direct execution toward an admissible final state

for now i guess we just let an operation have unit cost. can worry about cost models later

Our idea is to create a calculus founded on three operations:

Execute forward execution step

Rollback local rollback

Permute a jump to another branch that reflects the same history of operations (but differs in terms of the order of execution of the operations)

Once we recognized we fall into an inadmissible execution, our goal is to perform some operations in order to fall into an admissible execution preserving the observable behaviors. This leads us to two main concepts:

- What executions are admissible? (in the rest of the paper, we will adopt the classical idea of Software Transaction Memory)
- What behaviors of an execution are observable? (in the rest of the paper, we will consider final states as the observable behaviors of an execution)

We can plug in our approach different definitions of admissible executions, and observable behaviors.

In addition, we want to formalize Permute reaching a node in the tree that is at the same level of the current one. In this way, we do not rollback the execution, but indeed we advance it by one step as we would do if we were in an admissible execution. Intuitively, this means that our permute can be simulated by n rollbacks followed by m execution steps, with $m == n$[1].

Consider for instance the running example in Figure 7. Figure 8 depicts the tree of all possible executions of the running example. We consider the first and last execution as admissible, while the second one is not. This is a quite standard assumption for our running example: The two instructions of T1 conflict with the instruction of T2 (they all work on the same collection and on the same key). Therefore, the system should not allow T2 to execute its instruction if T1 has execute the first instruction but not the second one yet. This is usually achieved in two ways:

```
T1:
        put(k,v)
        get(k)

T2:
        remove(k)
```

Figure 7: The running example

treerunningexample.png

Figure 8: The tree of executions of the running example

- by synchronization mechanisms (e.g., locks) providing mutual exclusion between T1 and T2, or
- by rolling back the execution if the system falls into the second path and when executing remove.

Instead, we propose a novel approach. Consider the second path, and the situation in which we have already exposed history $[\mathtt{T1}:\mathtt{put},\mathtt{T2}:\mathtt{remove}]$. If we refrain from executing remove (that is, we replace it with a skip statement), we would apply a Permute step, that would yield us to the history $[\mathtt{T2}:\mathtt{remove},\mathtt{T1}:\mathtt{put}]$

In this way, the entire execution becomes $[\mathtt{T2}:\mathtt{remove},\mathtt{T1}:\mathtt{put},\mathtt{T1}:\mathtt{get}]$ that is an admissible execution.

Note that in contrast with Rollback and Execute, Permute requires an algebraic specification that permits efficient (bounded) editing of the state reflected by $[\mathtt{T1}:\mathtt{put},\mathtt{T2}:\mathtt{remove}]$ to arrive at $[\mathtt{T2}:\mathtt{remove},\mathtt{T1}:\mathtt{put}]$.

B. Permuting the execution – previous

The goal of our work is to try to *redirect* a bad execution into a good one by simulating the good trace up to that point. At a given point of an execution, we suppose that we can observe only two things:

1. where the execution of each transaction is. This can be formalized as $\mathit{transPP}(\tau) = [\mathtt{t} \mapsto \max(\mathit{indexes}(\mathtt{t},\tau)) : \mathtt{t} \in \mathsf{T}]$ where T contains all the identifiers of the transactions in the execution τ, and $\mathit{indexes}(\mathtt{t},\tau) = \{\mathsf{i} : \exists \sigma_j \rightarrow_{(\mathtt{t},\mathsf{i})} \sigma_{j+1} \in \tau\}$
2. what we can observe on the last state of the trace. The observational portion of the state is given by a function $\mathit{observe}(\sigma)$.

Given a bad trace $\sigma_0 \rightarrow \cdots \rightarrow \sigma_i$ we want to find a good trace $\sigma'_0 \rightarrow \cdots \rightarrow \sigma'_j$ such that

1. $\mathit{transPP}(\sigma_0 \rightarrow \cdots \rightarrow \sigma_i) = \mathit{transPP}(\sigma'_0 \rightarrow \cdots \rightarrow \sigma'_j)$
2. $\mathit{observe}(\sigma_0) = \mathit{observe}(\sigma'_0)$
3. $\mathit{observe}(\sigma'_j) = \mathit{observe}(f(\sigma_i))$ where f is a function that *adjusts* the state of the bad execution to fall into the good execution.

The two parameters of our framework are f (and a key component is how to compute it) and $\mathit{observe}(\sigma)$. We suppose that $\mathit{observe}(\sigma)$ returns the portion of the state that influences what is observable ”through” the semantics. This means that if we take two states equivalent modulo observability and we perform the semantics of a statement, we obtain two states that are equivalent modulo

[1] We discussed also the case in which $m > n$, but we are not yet sure if this case will be never interesting

observability. Formally,

$$\forall \sigma, \sigma' \in \Sigma : observe(\sigma) = observe(\sigma'), \forall \mathsf{s} \in \mathsf{St} : \langle \mathsf{s}, \sigma \rangle \rightarrow \sigma_1, \langle \mathsf{s}, \sigma' \rangle \rightarrow \sigma'_1$$
$$\Downarrow$$
$$observe(\sigma_1) = observe(\sigma'_1)$$

Based on the permutation function f we define the semantics $\mathbb{S}_{\mathbb{P}}[\![\mathsf{p}, \sigma_0]\!]$ that, if the execution falls into a bad trace, redirects the execution into a good trace by applying f.**Formalize it, not clear when exactly we apply the permutation**

We can now prove the soundness of our permutation. Intuitively, we prove that if one can observe only the observable part of the entry and the exit state (and not the intermediate state and the interleaving of transactions) it cannot notice the permutation we operate.

Theorem B.1 (Soundness of the permutation)**.**

$$\forall \sigma_0 \rightarrow \dots \rightarrow \sigma_i \in \mathbb{S}_{\mathbb{P}}[\![\mathsf{p}, \sigma_0]\!], \exists \sigma_0 \rightarrow \dots \rightarrow \sigma'_j \in \mathbb{S}[\![\mathsf{p}, \sigma_0]\!] : observe(\sigma_i) = observe(\sigma'_j)$$

Should we relax on the entry state using observe?

C. New ideas of the meeting on May, 2nd

C.1 Observational Equivalence

Instead of using the idea of an *observe* function and ask that the states are equal, we can rely on the observational equivalence relation between states. Another approach could be to adopt the POPL 02 Cousot and Cousot framework to define program transformations. First Pietro's intuition: "They deal with online and offline program transformation, and they define observational equivalence as an equivalence among abstractions of concrete executions. Indeed, what we are doing is slightly different: we perform a static analysis offline, and we change the semantics of the program (that - maybe - can be interpreted as a program transformation) online if we fall into a bad execution. On the other hand, I think that we could plug our work into their framework, and the main advantage would be to define the observational equivalence as an abstraction, and in particular as the abstraction we are performing in TVLA. In this way, we won't need to develop an ad-hoc definition of the correspondence between the observational equivalence and what we track with our static analysis."

C.2 Product of CFGs

We came up to the idea that in order to represent the interleaving of various threads. If we have two threads T1 and T2, we build up the product of all the nodes, and we add the edges that performs a step in the execution of one of the two threads. In this way we have some spurious paths (e.g., if we are inside a thread performing a loop and we perform one step in the other thread, also this step will be inside the loop), that we should refine through the abstract domain (e.g., adding program pointers of the various threads in the abstract state). In addition, we may have that a bad and a good trace are later joined. In order to distinguish between good and bad traces we will need to partition these two cases in the abstract domain (e.g., through a Bad abstraction predicate in TVLA).

Another idea from Eric was to use his PLDI 09 work (where they perform a sort of loop unrolling by expanding the CFG) but we didn't discuss it in the details.

D. Cartesian product of transactions

Given a tuple of n transactions $\mathsf{T} \in \mathsf{CFG}^n$, we build up the Cartesian product of the cfg of these transactions. Formally, we define the cfg of T by $\mathsf{cfg}_{\mathsf{T}} = \square_{(\mathsf{L}_i, \mathsf{E}_i) \in \mathsf{T}} (\mathsf{L}_i, \mathsf{E}_i)$ where $\square$ denotes the Cartesian product of graphs**Cite something?**. Formally,

$$\square_{(\mathsf{L}_i, \mathsf{E}_i) \in \mathsf{T}} (\mathsf{L}_i, \mathsf{E}_i) = (\mathsf{V}, \mathsf{E}) :$$
$$\mathsf{V} = \Pi_{(\mathsf{L}_i, \mathsf{E}_i) \in \mathsf{T}} \mathsf{L}_i$$
$$\mathsf{E} = \left\{ \begin{array}{l} (\mathsf{L}, \mathsf{L}', \mathtt{st}) : \mathsf{L}, \mathsf{L}' \in \mathsf{V} \wedge \\ \qquad \exists j \in [1..n] : \forall i \in [1..n] \setminus \{j\} : \mathsf{L}_i = \mathsf{L}'_i \wedge \\ \qquad \exists (\mathsf{L}_j, \mathsf{L}'_j, \mathtt{st}) \in \mathsf{E}_j \end{array} \right\}$$

The intuition behind the Cartesian product of cfgs is that (i) each node represents where the execution of each transaction is arrived, and (ii) each edge represents that one transaction performs the execution of an atomic step, while the others do not progress. In this way, we can rely on the Cartesian product of cfgs to compute the interleaving semantics.

Put the running example here

D.1 Semantics

On the Cartesian product of transactions we define the concrete semantics as follows:

$$\mathbb{S}_{\mathbb{CFG}}[\![\mathtt{cfg}_{\mathsf{p}}, \sigma_0]\!] = lfp^{\subseteq}_{(\mathsf{L}_0, \sigma_0)} \lambda \mathsf{T}. \mathsf{T} \cup$$
$$\left\{ \begin{array}{l} (\mathsf{L}_0, \sigma_0) \rightarrow \dots \rightarrow (\mathsf{L}_{i-1}, \sigma_{i-1}) \rightarrow (\mathsf{L}_i, \sigma_i) : \\ (\mathsf{L}_0, \sigma_0) \rightarrow \dots \rightarrow (\mathsf{L}_{i-1}, \sigma_{i-1}) \in \mathsf{T} \wedge \\ \exists (\mathsf{L}_{i-1}, \mathsf{L}_i, \mathtt{st}) \in \pi_2(\mathtt{cfg}_{\mathsf{p}}) \wedge (\sigma_{i-1}, \mathtt{st}) \rightarrow \sigma_i \end{array} \right\}$$

Lemma D.1 (Soundness of the concrete semantics on the Cartesian product of CFGs)**.** $\mathbb{S}[\![\mathsf{cfg}_{\mathsf{T}}, \sigma_0]\!] \subseteq \mathbb{S}_{\mathbb{CFG}}[\![\mathsf{cfg}_{\mathsf{T}}, \sigma_0]\!]$

In theory, they are equal and we can prove it, but maybe this is not interesting, maybe yes - it depends if we need an underapproximation for the good traces

D.2 Bad flows

Then we statically detect on the CFG of a set of transactions $\mathsf{cfg}_{\mathsf{T}}$ the flows that *may* lead to bad executions. Therefore, we build up a data flow analysis that tracks what actions may reach each label in $\mathsf{cfg}_{\mathsf{T}}$.

In particular, we are only interested in triples made by these elements since the properties we want to check are on the last three actions **We should enumerate the 4 cases and cite the work introducing them**. Therefore the abstract domain of our data-flow analysis is $(\mathsf{Lab} \cup \{\epsilon\})^3$, where ϵ represents situation in which we have less than three elements. Our data flow analysis is forward and possible. Therefore, our analysis is defined by the following equations:

$$In(\mathsf{l}) = \bigcup_{(\mathsf{l}', \mathsf{l}) \in \mathsf{e}} Out(\mathsf{l}')$$
$$Out(\mathsf{l}) = Gen(\mathsf{l}) \setminus Kill(\mathsf{l})$$
$$Kill(\mathsf{l}) = \{(\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3) : (\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3) \in \mathsf{Lab}^3\}$$
$$Gen(\mathsf{l}) = \{(\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3) : \exists \mathsf{a}_4 \in \mathsf{Act} \cup \{\epsilon\} : (\mathsf{a}_2, \mathsf{a}_3, \mathsf{a}_4) \in In(\mathsf{l}) \wedge$$
$$\mathsf{a}_1 \in getAct(getWeigth(\mathsf{l}), \})$$

where e represents the edges of the cfg, and $getAct(\mathtt{st}, TODO)$ returns the set of actions (r and/or w) the given statement may perform.**Since we have the statements of the edges and not in the nodes, the getLabel function is somewhat broken. I should add the out label and the transaction that performs the statement**

Finally, we tag the edges that could expose bad behaviors. For each edge

In particular, for each edge in $\mathsf{cfg}_{\mathsf{T}}$ we consider all the triples of actions generated by this edge. If at least one of these triples represents a conflict, we tag the edge as bad.**Note: we should have one triple per key, but since we are assuming to have only one key statically known we consider this case. This sounds weak** Bad edges are aimed at over-approximating bad executions.

Theorem D.2. $\forall \tau \in \mathsf{BadTraces} \cap \mathbb{S}[\![\mathsf{p}, \sigma_0]\!] : \tau \in \mathbb{S}_{\mathbb{C}}[\![\mathsf{cfg}_{\mathsf{T}}, \sigma_0]\!] \cap \mathsf{BadTraces}$

We should define exactly the semantics over the cartesian product of cfgs and show how the trace is built from there

XXXXXXX TO BE REMOVED!!! XXXXXXX

E. Concrete domain and semantics

Our concrete domain is a standard finite trace domain. We define by $\mathsf{A}^{\mp}$ the set of all finite traces of elements in A.

Let Σ_{m} be the set of concrete states that represent both the shared and the internal memory state of the system. We suppose that an atomic small-step semantics $\rightarrow: \Sigma_{\mathsf{m}} \times \mathsf{St} \rightarrow \Sigma_{\mathsf{m}}$ is provided. Given n transactions, the definition of our concrete trace semantics relies on labeled states $\Sigma = \mathsf{Lab}^n \times \Sigma_{\mathsf{m}}$. Based on this semantics, we define the program semantics as follows:

$$\mathbb{S}[\![\mathtt{p}, \sigma_0]\!] = lfp^{\subseteq}_{(\mathsf{L}_0, \sigma_0)} \lambda \mathsf{T}.\mathsf{T} \cup \left\{ \begin{array}{l} \{(\mathsf{L}_0, \sigma_0) \rightarrow \cdots \rightarrow (\mathsf{L}_{i-1}, \sigma_{i-1}) \rightarrow (\mathsf{L}_i, \sigma_i) : \\ \quad (\mathsf{L}_0, \sigma_0) \rightarrow \cdots \rightarrow (\mathsf{L}_{i-1}, \sigma_{i-1}) \in \mathsf{T} \wedge \\ \quad \mathtt{t} \in dom(\mathtt{p}) \wedge \exists \mathsf{l} \in \mathsf{Lab}, \mathtt{st} \in \mathsf{St} : (\pi_{\mathtt{t}}(\mathsf{L}_{i-1}), \mathsf{l}, \mathtt{st}) \in \pi_2(\mathtt{p}(\mathtt{t})) \wedge \\ \quad \mathsf{L}_i = rep(\mathsf{L}_{i-1}, \mathtt{t}, \mathsf{l}) \wedge (\sigma_{i-1}, \mathtt{st}) \rightarrow \sigma_i \end{array} \right\}$$

where $\mathsf{L}_0 = \mathtt{entry}^{|dom(\mathtt{p})|}$ and $rep(\mathtt{t}, i, v)$ replaces the i-th component of the tuple $\mathtt{t}$ with value v.

E.1 Property

We distinguish between *good* and *bad* executions by looking to the interleaving of the execution of different transactions **Probably we should add something about the state (e.g., to check that two statements interfer on the same key). This is just an abstraction of the trace.**.

An action performed by a transaction i on a key $\mathtt{k}$ is represented by $a_i(\mathtt{k}) \in \mathsf{Act}$. We distinguish between read and write actions represented by r and w, respectively. Statements in $\mathtt{s}$ by transaction i are interpreted as read and/or write as follows:

- $\mathtt{m.put(k, v)}$ is represented by $w^i(\mathtt{k})$,
- $\mathtt{x = m.get(k)}$ is represented by $r^i(\mathtt{k})$,
- $\mathtt{m.remove(k)}$ is represented by $w^i(\mathtt{k})$, and
- $\mathtt{x = m.putIfAbsent(k, v, v')}$ is represented by $r^i(\mathtt{k})$ (if $\mathtt{k}$ is already in the map) *and* $w^i(\mathtt{k})$ (otherwise).

Given a statement $\mathtt{st}$ and a transaction k, we define by $getAct(\mathtt{st}, k)$ the function that returns the corresponding action.

We then define a function $\pi_{\mathsf{Lab}}(\mathtt{p}, \tau)$ that projects a trace of states τ produced by a program $\mathtt{p}$ to the sequence of actions it performed[2]:

$$\begin{array}{c} \pi_{\mathsf{Lab}}(\mathtt{p}, (\mathsf{L}_0, \sigma_0) \rightarrow \cdots \rightarrow (\mathsf{L}_n, \sigma_n)) = \{\mathsf{a}_1 \rightarrow \cdots \rightarrow \mathsf{a}_n : \\ \forall j \in [1..n] : extractStep(\mathtt{p}, \mathsf{L}_{j-1}, \mathsf{L}_j) = (\mathtt{t}, k) \wedge \\ getAct(\mathtt{t}, k) = \mathsf{a}_j\} \end{array}$$

where $extractStep(\mathtt{p}, \mathsf{L}, \mathsf{L}')$ given two labels representing a step of the execution and the program returns the statement and the transaction that performed the step.

We need to build a function $isBad(\mathsf{a}_1 \rightarrow \cdots \rightarrow \mathsf{a}_n)$ returns $\mathtt{true}$ if and only if the given sequence of actions represents a serializability violations.**This is not quite right, revise it later**

Finally, given a program $\mathtt{p}$ and an initial state σ_0, we can partition the set of traces into good and bad traces.

$$\begin{array}{l} \mathsf{T} = \mathbb{S}[\![\mathtt{p}, \sigma_0]\!] \\ \mathsf{BadTraces} = \{\tau \in \mathsf{T} : isBad(\pi_{\mathsf{Lab}}(\tau))\} \\ \mathsf{GoodTraces} = \mathsf{T} \cap \mathsf{BadTraces} \end{array}$$

E.2 Abstract domain and semantics

As usual in abstract interpretation, we approximate the concrete domain and semantics with an abstract domain and semantics. The abstract domain forms a Galois connection with the concrete domain, while the abstract semantics approximates the concrete one. **The abstract semantics should be tuned at the level of traces, so we can present the Cartesian product as an abstraction of the original traces**

F. Warping system

F.1 Observational Equivalence

We assume that observations on the state can only be made by state transformers. We denote $\sigma \xrightarrow{m/k} \sigma'$ a state transition where some method m has been invoked, and value k returned (we leave the domains of m and k undefined for now).

For two states σ_1 and σ_2, observational equivalence is defined as follows:

$$\sigma_1 \sim \sigma_2 \quad \Leftrightarrow \quad \begin{array}{rl} & \sigma_1 \xrightarrow{m/k_1} \sigma_1' \text{ implies that} \\ (i) & \exists k_2\, \sigma_2'.\ \sigma_2 \xrightarrow{m/k_2} \sigma_2' \\ (ii) & \forall k_2\, \sigma_2' \text{ such that } \sigma_2 \xrightarrow{m/k_2} \sigma_2'. \\ & \sigma_2 \xrightarrow{m/k_2} \sigma_2' \ \wedge\ k_1 = k_2 \end{array}$$

F.2 Trace Warping

Consider some bad trace $\tau = \sigma_0 \rightarrow \cdots \rightarrow \sigma_i$. We want to find a good trace $\tau' = \sigma_0' \rightarrow \cdots \rightarrow \sigma_j'$ such that $\sigma_i \sim \sigma_j'$. We do this via a *warp* function f, which adjusts the state of the bad execution to fall into the good execution.

For $\tau = \sigma_0 \rightarrow \cdots \rightarrow \sigma_i \in \mathsf{BadTraces}$, and $\tau' = \sigma_0' \rightarrow \cdots \rightarrow \sigma_j'$,

$$\mathsf{warp}(\sigma_i, f(\sigma_i)) \quad \Leftrightarrow \quad f(\sigma_i) \sim \sigma_j' \ \wedge\ \sigma_0 \sim \sigma_0' \ \wedge\ \tau' \notin \mathsf{BadTraces}$$

consider not including $\mathsf{BadTraces}$ **in the def of warp. maybe easier to permit warping to other bad traces**

F.3 Abstract Trace Warping

define hat sim

We now lift to the abstract domain. Consider some bad trace $\tau = \hat{\sigma}_0 \rightarrow \cdots \rightarrow \hat{\sigma}_i$. We want to find a good trace $\tau' = \hat{\sigma}_0' \rightarrow \cdots \rightarrow \hat{\sigma}_j'$ such that $\hat{\sigma}_i \hat{\sim} \hat{\sigma}_j'$. We do this via an *abstract warp* function $\hat{f}$, which adjusts the state of the bad execution to fall into the good execution.

For $\tau = \hat{\sigma}_0 \rightarrow \cdots \rightarrow \hat{\sigma}_i \in \mathsf{BadTraces}$, and $\tau' = \hat{\sigma}_0' \rightarrow \cdots \rightarrow \hat{\sigma}_j'$,

$$\mathsf{warp}(\hat{\sigma}_i, \hat{f}(\hat{\sigma}_i)) \quad \Leftrightarrow \quad \hat{f}(\hat{\sigma}_i) \hat{\sim} \hat{\sigma}_j' \ \wedge\ \hat{\sigma}_0 \hat{\sim} \hat{\sigma}_0' \ \wedge\ \tau' \notin \mathsf{BadTraces}$$

G. TVLA-based analysis

We instantiated this framework with a TVLA based analysis. We adopted the following representation:

- we represent through a summary node what is inside the map at the beginning of the execution
- for all the keys and the values that are parameters of the two transactions, we represent them with concrete nodes
- we adopt a binary predicate r to link the map to the summary node
- we adopt a binary predicate k to link a parameter key to the map to represent that the key is in the map (note that this predicate is always 0 or 1, never half!)
- we adopt a binary predicate val to represent that a key is connected to a value. This val connects the summary node to

[2] In the formal definition we ignore execution steps that do not produce actions (i.e., the execution of statements in $\mathtt{c}$)

itself to represent the initial values stored in the map, and it connects the concrete node of a parameter key to a concrete node of a parameter value if the key is insider the map and it is related to the given value

We then represent various entry state in which the predicate k (and val between parameter keys and parameters values) is 1 or 0 to represent all the possible combination when the entry map contains or not a parameter key (and the parameter key is related or not to a parameter value).

Given two abstract states of our analysis, we know that all their concretizations are observationally equivalent.

Theorem G.1. $\forall \sigma_1, \sigma_2 \in \gamma(\overline{\sigma}) : \sigma_1 \sim \sigma_2$

H. Instance of the warping system

Abstract domain $\overline{\Sigma}^{\sharp} = \wp(\overline{\Sigma} \times \{\mathsf{good}, \mathsf{bad}\})$

$\mathbb{Q}$ finite set of queries (with parameters)

This set of queries represents the set of observations we can make on a state of the execution (e.g., check if a key is in the map, get the value stored in a given key, ...). This means that we consider that two concrete states are observationally equivalent if they give the same answers to all these queries (Property 1).

First assumption: we have an $eval$ function (that evaluates a query) such that $\forall \mathsf{q} \in \mathbb{Q}, \overline{\sigma} \in \overline{\Sigma} : \overline{eval}(\mathsf{q}, \overline{\sigma}) \in \{\mathsf{true}, \mathsf{false}\}$. In particular, this means that we get precise answers to the queries (e.g., the key is / is not in a map), since we can answer only true or false, and not $\top$. By Property 1 we have then that all the concrete states concretized from an abstract state $\overline{\sigma} \in \overline{\Sigma}$ they belong to the same class. In fact, if we get as answer to a query on the abstract state true (or false), we get true (or false) on all the possible concretizations of $\overline{\sigma}$.

Second assumption: we have a function $\Delta : \overline{\Sigma} \times \overline{\Sigma} \rightarrow \mathsf{St}^* \cup \{\top\}$. This function, given two states, returns the set of statements you have to execute to go from the first abstract state to the second one. Formally, $\overline{\mathbb{S}}[\![\Delta(\overline{\sigma}, \overline{\sigma}'), \overline{\sigma}]\!] = \overline{\sigma}'$ if $\Delta(\overline{\sigma}, \overline{\sigma}') \neq \top$.

Then we have an entry state made by good abstract states, we take the Cartesian product of the cfgs, and we compute entry and exit state per node in the cfg. We need to explain (1) how we assign good and bad, and (2) how we prove the correlation between entry and exit state preserving the concrete semantics.