Using Abstract Interpretation to Correct Synchronization Faults

Pietro Ferrara[1]*, Omer Tripp[2]*, Peng Liu[3], and Eric Koskinen[4]**

[1] Julia Srl, Italy
pietro.ferrara@juliasoft.com
[2] Google Inc., U.S.A.
trippo@google.com
[3] IBM T.J. Watson Research Center, U.S.A.
liup@us.ibm.com
[4] Yale University, U.S.A.
eric.koskinen@yale.edu

Abstract. We describe a novel use of abstract interpretation in which the abstract domain informs a runtime system to correct synchronization failures. To this end, we first introduce a novel synchronization paradigm, dubbed *corrective synchronization*, that is a generalization of existing approaches to ensuring serializability. Specifically, the correctness of multi-threaded execution need not be enforced by previous methods that either reduce parallelism (pessimistic) or roll back illegal thread interleavings (optimistic); instead inadmissible states can be altered into admissible ones. In this way, the effects of inadmissible interleavings can be compensated for by modifying the program state as a transaction completes, while accounting for the behavior of concurrent transactions. We have proved that corrective synchronization is serializable and give conditions under which progress is ensured. Next, we describe an abstract interpretation that is able to compute these valid serializable post-states w.r.t. a transaction's entry state by computing an under-approximation of the serializable intermediate (or final) states as the fixpoint solution over an inter-procedural control-flow graph. These abstract states inform a runtime system that is able to perform state correction dynamically. We have instantiated this setup to clients of a Java-like Concurrent Map data structure to ensure safe composition of map operations. Finally, we report early encouraging results that the approach competes with or out-performs previous pessimistic or optimistic approaches.

1 Introduction

Concurrency control is a hard problem. While some thread interleavings are admissible (if they involve disjoint memory accesses), there are certain interleaving scenarios that must be inhibited to ensure serializability. The goal is to automatically detect, with high precision and low overhead, the inadmissible interleavings, and avoid them.

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

---

* The research leading to this paper was conducted while the author was at IBM Research.
** The research was supported by NSF CCF Award #1421126 and conducted partially while the author was at IBM Research.

- *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 executions has gone away. Locks, mutexes, semaphores, and some transactional memory (TM) implementations [11,18] are all examples of how to enforce mutual exclusion, or pessimistic synchronization.
- *Optimistic synchronization*: As an alternative to pro-active (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 abort-like remediation steps are taken. Many TM implementations operate this way [12], logging memory accesses, aborting transactions, and reversing the effects.

The pessimistic approach is useful if critical sections are short, there is little available concurrency, and the involved memory locations are well known [13]. 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 [16]. In both of these cases, however, the concurrency paradigm is designed to exploit domain-specific windows of opportunity where there is a low amount of conflict. These are, in a sense, low-hanging fruit, and there are many other situations of practical interest where there is unavoidable contention. (We will give a simple example in the next section.) Neither of these existing approaches offer a way to tackle contention head-on.

Corrective Synchronization. In this paper, we take a first step in formulating and exploring a novel synchronization paradigm that generalizes both the pessimistic and optimistic approaches. In our approach, dubbed *corrective synchronization*, conflicting transactions may begin to execute concurrently (unlike pessimism), yet when conflict occurs, the remediation is not simply to abort (unlike optimism). Instead, a thread resolves contention by dynamically *altering* the inadmissible state into an acceptable one, accounting for the behavior of concurrent threads, so as to guarantee serializability.

This paper. Corrective synchronization, as a concept, opens up a vast space of possibilities for concrete synchronization protocols. In this paper, we take a first step in exploring this space with a formalism, proof of serializability, and a novel use of abstract interpretation and dynamic instrumentation. The key idea can be illustrated with our $\mathsf{corr}\ t$ proof rule:

$$\frac{\Gamma, (t, s) \vdash s \to^* (T, \mu', \sigma, L)}{\Gamma, (t, s) \vdash (T, \mu, \sigma, L) \to (T, \mu[t \mapsto \mu'(t)], \sigma, L)}\ \mathsf{corr}\ t$$

As is typical, we model transactions as a transition system where a state configuration s consists of tuples (T, μ, σ, L). Here T is a set of transaction identifiers, μ is a mapping from transaction id to a thread-local replica of the state, σ is the shared state and, following [15], we use a shared log of events L to track the effects of committed transactions. For our purposes, we include a context Γ which maps each transaction id t to the configuration just before the corresponding transaction began.

The key idea here is thread-local correction, whereby a single thread t can apply a correction by jumping from μ to $\mu[t \mapsto \mu'(t)]$. Thread t is permitted to do so, provided that there was an *alternate execution path* $s \to^* (T, \mu', \sigma, L)$ from the configuration s in which t began to some other (T, μ', σ, L). After a correction, the thread may be able to perform a commit, which (logically) involves replaying the mutation on the shared state. This rule is fairly simple yet has significant consequences and, to the best of our knowledge, nothing like this exists in the literature. One can think of pessimistic synchronization as an almost trivial restriction in which conflicting executions never occur and this rule is never needed. Meanwhile, optimistic synchronization permits only corrections back to thread t's initial configuration. For the purposes of this paper, the alternate path to μ' must be under the same set of concurrent transactions T, though this restriction can be relaxed, which we leave for future work. We have proved that our definition of corrective synchronization is serializable (Section 3) and provide conditions under which progress is guaranteed.

Challenges & Contributions. With definitions and serializability established, we move on to two key challenges that pertain to realizing corrective synchronization:

- How do we compute each thread's alternative (serializable) post-states?
- Given an incorrect state, how do we efficiently recover to a correct post-state?

For the sake of concreteness, we focus on concurrent Java-like programs whose shared state is encoded as one or more ConcurrentMap instances. We tackle the above challenges via a novel use of abstract interpretation, equipped with a specialized abstraction for maps, to derive the correct post-states, or "targets", in relation to a given pre-state. Our abstract interpretation computes an under-approximation of the serializable intermediate (or final) states as the fixpoint solution over an inter-procedural control-flow graph. We prove that the computed target states are progress-safe, i.e. the system is not in a stuck state after a correction. We then show how these target states can be used by a runtime system to dynamically correct an execution and jump to a target state.

In summary, this paper makes the following principal contributions. (1) We present an alternative to both the pessimistic and optimistic synchronization paradigms, dubbed *corrective synchronization*, whereby serializability is achieved neither via mutual exclusion nor via rollbacks, but through correction of the post-state according to a relational pre-/post-states specification. (2) We provide a formal description of corrective synchronization. This includes soundness and progress proofs as well as a clear statement of limitations. (3) We have developed an abstract interpretation to derive the prestate/post-states specification for programs that encode the shared state as one or more concurrent maps. (4) We have developed a runtime system that is able to use pre/post-state specifications to correct behaviors dynamically. (5) We report encouraging preliminary experimental results on a prototype implementation.

2 Technical Overview

We now walk the reader through a high-level overview of corrective synchronization with an example. We describe the conceptual details at a technical level, and then the two main algorithmic steps.

Running Example. As an illustrative example, we refer to the code fragment on Figure 1, where a shared Map object, (pointed-to by) map, is manipulated by method updateReservation.

Let us assume that different threads invoking this method are all attempting to simultaneously move reservations into the same nslot. Doing so optimistically would lead to multiple rollbacks (even under boosted conflict detection [11], since the operations due to different threads do not commute), and thus poor performance. Pessimistic mutual exclusion, on the other hand, would block all but one thread until the operation completes, which is far from optimal if new Res() is an expensive operation.

```
Res updateReservation(Slot oslot, Slot nslot) {
  Res r = map.remove(oslot);
  if (r == null)
    r = new Res();
  Res dead = map.replace(nslot, r);
  return dead;
}
```

Fig. 1: Example that updates reservation for customers, moving an entry from one date/time slot to another.

Conceptual Approach. By *corrective synchronization* we mean the ability to transform a concurrent run that, in its present condition, may not be serializable into a run that is serializable. Stated formally, corrective synchronization is a relationship $h \sim h'$ between histories, such that (i) h and h' share the same initial state, and (ii) h and h' share the same log of committed operations (i.e., they agree on the operations on the shared-state). One can think of h' as an alternate parallel reality to h.

The first condition ensures that corrective synchronization yields a feasible outcome. The second is the requirement not to roll back updates to the shared state. These two conditions distinguish corrective synchronization from existing solutions: Unlike pessimistic approaches, bad behaviors may occur under corrective synchronization. That is, they are not avoided, but handled as they manifest. Unlike optimistic solutions, the core handling mechanism is not to retry the transaction (or parts thereof), which implies rolling back (either committed or uncommitted) updates to the shared log, but rather to "warp" to another state. To illustrate our approach, consider the following history:

$$
\begin{array}{lll}
[& t_1\text{: remove}(10{:}00)/\alpha & \rightarrow t_2\text{: remove}(13{:}00)/\text{null} \\
\hookrightarrow & t_1\text{: if}(\ldots) & \rightarrow t_2\text{: if}(\ldots)\ \text{new Res}()/\delta \\
\hookrightarrow & & \quad t_2\text{: replace}(11{:}00)/\beta \\
\hookrightarrow & t_1\text{: replace}(11{:}00)/\delta & \\
\hookrightarrow & t_1\text{: return } \delta & \rightarrow t_2\text{: return } \beta \quad]
\end{array}
$$

Each step in this history is labeled with the transaction identifier (t_1 or t_2) and the code that the transaction is executing. With this history, $11{:}00$ is associated with β in the final state of the map. This history is clearly not serializable. t_2 cannot return β when $11{:}00$ is associated with β in the final state of the map. However, we can warp to a history that is serializable by applying a correction. In this case, note that there are two possible serializable histories: (i) t_1 goes first and returns β, t_2 goes second and returns α, and in the final state $11{:}00$ is mapped to δ; (ii) t_2 goes first and returns β, t_1 goes second and returns δ, and in the final state $11{:}00$ is mapped to α. For the sake of efficiency, we can correct the above execution by chosing serial history (ii) as the target state and the correction is quick and easy: update the map so that $11{:}00$ is associated with α. Using (i) as a target is also possible, but would have required a correction to the map as well

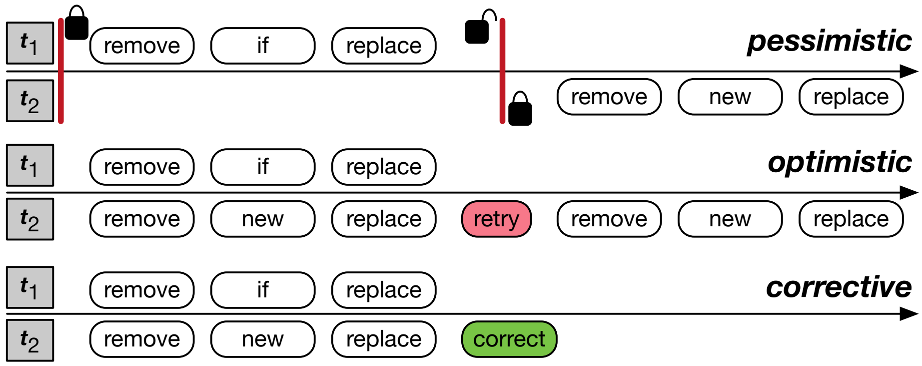

Fig. 2: Interleaved execution of two instances of updateReservation using pessimistic, optimistic, and corrective synchronization.

as the return values of t_1 and t_2. Note that the corrective actions above are of a general form, which is not limited to two threads. For any number of threads, the corrected state would have one privileged thread deciding the return value (i.e., the value of dead) for all threads as well as what $11\!:\!00$ should be associated with in the map. Also note that the correction does *not* directly modify the shared state; rather, the correction is made to a thread-local replica of the state. After the correction, if the thread is able to commit, then shared-state mutations are applied at commit time.

How does this corrective approach compare to handling of the situation by pessimistic or optimistic approaches? We illustrate the difference between corrective synchronization and classic optimistic and pessimistic synchronization in Figure 2. We visually represent concurrent execution of two instances of the updateReservation code using pessimistic locks, optimistic TM and corrective synchronization (proceeding horizontally left-to-right). Pessimism serializes execution, and so there is no performance gain whatsoever. As for optimistic and corrective synchronization, we consider the interleaving scenario specified above. Both optimistic and corrective synchronization, allowing the problematic chain of interleavings, reach a nonserializable state. Optimism resolves this by retrying the entire transaction executed by, say, thread t_2. This yields serial execution, similar to the pessimistic run, where t_2 runs after t_1. Corrective synchronization, instead, “fixes” the final state, allowing t_2 to complete without rerunning any or all of its code. Our experiments suggest the corrective actions are — relatively speaking — inexpensive, especially compared to the alternatives of either blocking or aborting/restarting all threads but one.

We refer to corrective synchronization as *sound* if h' is the prefix of a serializable execution of the system. We refer to corrective synchronization as *complete* if for any h, all the h's that satisfy the conditions above are in the relation $\sim$. In the rest of this paper, we describe our method of computing a sound yet incomplete set of corrective targets via static analysis of the concurrent library. A solution that is not complete faces the possibility of stuck runs: Given a (potentially) nonserializable execution prefix, the system does not have a corresponding serializable prefix to transition to. In this paper, we do not present a solution to the completeness problem, which we leave as future work. In the meantime, there are two simple strategies to tackle this problem: (i) *manual* *specification*, whereby the user completes the set of corrective targets to ensure that there are no stuck runs (in our implementation, the targets are computed offline via static analysis, letting the user complete the specification ahead of deployment); and (ii) *complementary techniques*, such as optimism, which the system can default to in the absence of a corrective target.

Computing Corrective Targets. A simpler and more abstract specification to work with, compared to complete execution prefixes, is triplets (s, s', s'') of states, such that there exist prefixes h and h' as above with respective initial and current states (s, s') and (s, s''), respectively. This form of specification is advantageous, because the corresponding runtime instrumentation is minimal compared to tracking traces. At the same time, however, initial and current states are a strict abstraction of complete traces, and so they do not point back to prefixes h and h'.

Mapping back from pairs of states to prefixes requires an oracle. In our work we compute the oracle as a relational abstract interpretation solution over the program that is sound yet incomplete. Specifically, an underapproximation of the serializable intermediate (or final) states is computed as the fixpoint solution over an interprocedural control-flow graph (CFG) of the form: $t_1 \rightarrow t^{\star}_{2 \ldots n} \rightarrow t_{n+1} \rightarrow t'_1 \rightarrow t'^{\star}_{2 \ldots n} \rightarrow t'_{n+1} \rightarrow \ldots$, where t, t', etc denote different transaction types (i.e., transactions executing different code), and n is unbounded, simulating a nondeterministic loop. This representation simulates an unbounded number of instances of transactions that are executed sequentially.

We go into detail about this representation in Section 5, but here note that (i) this representation reflects the effects of serial execution of the transactions, and so the corrective targets are guaranteed to be sound; (ii) the nondeterministic loop captures an unbounded number of transactions; and (iii) the first and last transactions of a given type are purposely disambiguated to boost the precision of static analysis over the simulated execution. As an illustration of the third point, in our running example the first transaction t_1 is modeled precisely in inserting the key/value pair into the Map object. Analogously, the last transaction t_{n+1} can be confirmed not to update the key/value mapping.

Runtime Synchronization. The runtime system has two main responsibilities. First, it must track whether an execution has reached a (potentially) bad state. Second, if such a state arises, then the runtime system must map the current state onto a state that shares the same initial state and is known, by the oracle, to have a serializable continuation. We address the first challenge via a coarse conflict-detection algorithm that tracks API-level read/write behaviors (at the level of Map operations). If read/write or write/write conflicts arise, then corrective synchronization is triggered in response. If the oracle was not able to compute a target for the correction (e.g., because of a loss of precision of the static analysis), then our approach can apply optimistic synchronization

We expand on the above challenges in Section 5, and provide encouraging experimental results on a simple prototype in Section 6. Prior to that, in Section 4, we provide a formal statement of corrective synchronization.

3 Semantics of Corrective Synchronization

We now introduce a generic transaction semantics for corrective synchronization. We prove soundness (serializability), and give conditions under which progress is ensured.

Notation. We use the following semantic domains:

$$
\begin{array}{rlrl}
c \in \mathcal{C} := & \text{command} & t \in T \subset \mathcal{T} := & \text{transaction IDs} \\
\sigma \in \Sigma := & \text{shared state} & \sigma_t := & \text{local state of } t \\
L \in \mathcal{L} := & \text{shared log} & L_t := & \text{local log of } t
\end{array}
$$

$$s = (T, [t \mapsto (c_t, \sigma_t, L_t)]_{t \in T}, \sigma, L) \in \mathcal{S} := \text{system state}$$

Following [15], our semantics uses local logs to track local operations, and shared logs to track committed operations. We assume that the set Σ of shared states is closed under composition, denoted $\cdot$. That is, $\forall \sigma, \sigma' \in \Sigma.\ \sigma \cdot \sigma' \in \Sigma$. Hence, we can decompose a given shared state into (disjoint) substates (the standard decomposition being into memory locations), such that we can easily refer to the read/write effects of a given operation. For that, we additionally define two helper functions, $r, w\colon\ \mathcal{C} \times \mathcal{S} \rightharpoonup \Sigma$, such that r (*resp.* w) computes the portion of the shared state read (*resp.* written) by a given atomic operation. The notation $\rightharpoonup$ denotes that w and r are partial functions. The shared log L consists of pairs $\langle t, o \rangle$, where t is a transaction identifier, o is the operation executed by t, and $w(c) \neq \bot$.

Transition System. Execution of the transition system is represented by five events:

$$\textsf{bgn}\ t \quad \frac{t \notin T}{\Gamma \cup (t, (T, \mu, \sigma, L)) \vdash (T, \mu, \sigma, L) \to (T \cup \{t\}, \mu \cdot [t \mapsto (c, \bot, \epsilon)], \sigma, L)}$$

$$\textsf{local}\ t \quad \frac{t \in T, \mathbb{C}[\![c_t, \sigma_t]\!] = (c'_t, \sigma'_t)}{\Gamma \vdash (T, \mu, \sigma, L) \to (T, \mu[t \mapsto (c'_t, \sigma'_t, L_t)], \sigma, L)}$$

$$\textsf{cmt}\ t \quad \frac{t \in T, L_t \neq \epsilon, \textsf{serpref}\ L \cdot L_t}{\Gamma \vdash (T, \mu, \sigma, L) \to (T, \mu[t \mapsto (c_t, \sigma_t, \epsilon)], [\![L_t]\!](\sigma), L \cdot L_t)}$$

$$\textsf{end}\ t \quad \frac{t \in T, \mu(t) = (\textsf{skip}, \_, \epsilon)}{\Gamma \vdash (T, \mu, \sigma, L) \to (T \setminus \{t\}, \mu \setminus [t \mapsto \mu(t)], \sigma, L)}$$

$$\textsf{corr}\ t \quad \frac{t \in T, s \rightsquigarrow (T, \mu', \sigma, L)}{\Gamma, (t, s) \vdash (T, \mu, \sigma, L) \to (T, \mu[t \mapsto \mu'(t)], \sigma, L)}$$

The bgn event marks the beginning of a transaction. In this and all rules, we work with a context Γ, consisting of pairs (t, s) that correlate transaction identifier t with the state configuration s that immediately preceded t's start, captured in the bgn event. During its execution, a transaction modifies only its local state and local log L_t, as seen in the local event rule. Here t's corresponding local configuration is denoted (c_t, σ_t, L_t) and $\mathbb{C}$ represents the transition relation for local operations. The cmt event fires when a transaction publishes its outstanding log of operations that affect the shared state to the shared state and log. We use helper function $\textsf{serpref}\colon\ \mathcal{L} \to \{\textsf{true}, \textsf{false}\}$ to (conservatively) determine whether a given shared log is the prefix of some serializable execution log. The end event marks the termination of a transaction.

Corrective action occurs in the corr event. This event enables a transaction to modify its local state and log, under certain restrictions, as a means to recover from potentially inadmissible thread interleavings. Note that the corr rule only applies changes to t's local configuration. All other transactions retain their original local configurations. The intuition is that corr *corrects* the execution by jumping transaction t to a state that is reachable starting from the entry state through a serialized execution.

Theorem 1 (Soundness). *A terminating execution of the transition system yields a serializable shared log (history).*

Proof Sketch. The cmt *event acts as a gatekeeper, demanding that the log prefix $L \cdot L_t$ including the outstanding events about to be committed is serializable. The check executes atomically together with the log update. Hence the system is guaranteed to terminate with a serializable shared log.*

Definition 1 (Progress). *We say that the transition system has made progress, transitioning from (global) state s to (global) state s', if the associated event e for $s \xrightarrow{e} s'$ is either a* cmt *event or an* end *event.*

Definition 2 (Progress-safe corrective synchronization). *Let* corr t *occur at system state $s = (T, \mu, \sigma, L)$, such that state $s' = (T, \mu[t \mapsto (c_t, \sigma_t, L_t)], \sigma, L)$ is reached. Assume that there is a reduction $(\sigma_t, c_t, L_t) \longrightarrow (\sigma'_t, c'_t, L'_t)$, such that at system state $s'' = (T, \mu[t \mapsto (\sigma'_t, c'_t, L'_t)], \sigma, L)$ either (i)* cmt t *is enabled or (ii)* end t *is enabled. Then we refer to* corr t *at s with target (σ_t, c_t, L_t) as* progress safe.

From the perspective of transaction t, the local states of other transactions are irrelevant to whether a commit (or end) transition is enabled for t. The only cause of a failed commit is if other threads have committed. We can therefore relax the definition above to refer to any system state $s'' = (T', \mu', \sigma, L)$, such that $t \in T'$ and $[t \mapsto (\sigma'_t, c'_t, L'_t)] \in \mu'$.

Given transaction t with local state (σ_t, c_t, L_t), we refer to target (σ'_t, c_t, L'_t) as a *self-corrective target*. After corrective synchronization, the transaction has the same command left to reduce, but its state and outstanding log of operations are modified. A specific instance is $(\sigma_t, \mathsf{skip}, L_t) \rightarrow (\sigma'_t, \mathsf{skip}, L'_t)$. This pattern of corrective synchronization is progress safe if commits are attempted at join points, which enables simulation of alternative control-flow paths (and therefore also logged effects) via corrective synchronization.

Theorem 2 (Progress). *If (i) a* corr t *event only fires when a transaction t reaches a commit point but fails to commit, and (ii) corrective synchronization instances are progress safe, then progress is guaranteed.*

Proof Sketch. Given system state s, if there exists a transaction t that is able to either commit or complete then the proof is done. Otherwise, there is a transaction t that reaches a commit point at some state s' and fails. At this point, corr t *is the only enabled transition for t, and by assumption (ii), the corrective synchronization instance is progress safe. At this point, there are two possibilities. Either t proceeds without other threads modifying the shared state, such that a commit or completion point is reached by t (without corrective synchronization prior to reaching such a point according to assumption (i)), in which case progress has been achieved, or one or more threads interfere with t by committing their effects, in which case too progress has been achieved.*

Definition 3 (Complete corrective synchronization). *We say that the system is complete w.r.t. corrective synchronization if for any state s, if a $\mathsf{corr}\ t$ transition is executed in s, then the selected corrective target satisfies progress safety.*

Lemma 1 (Termination). *Assume that the system performs corrective synchronization only on failed commits, and is complete w.r.t. corrective synchronization. Then for any run of the system where finitely many transactions are created, each having only finite serial execution traces, termination is guaranteed.*

Proof Sketch. *The first two assumptions guarantee progress, as established above in Theorem 2. Since transactions are finite, each transaction may perform finitely many cmt transitions before terminating via an end transition. This implies that after finitely many transitions, some transaction t will terminate. This argument applies to the resulting system until no transactions are left.*

4 Thread Local Semantics

We now instantiate the theoretical framework introduced in Section 3 to a language supporting some standard operations on concurrent shared maps. We define the thread-local concrete semantics of this language instantiating $\mathbb{C}$ of the local rule. Following standard abstract interpretation theory, we then introduce an abstract domain and semantics that computes an approximation of the concrete semantics. This thread-local abstract semantics will be used in Section 5 to compute progress-safe corrective "targets".

Language. We focus our formalization on the following language fragment:

$$
\begin{aligned}
\mathtt{s} ::= &\ \mathtt{m.put(k, v)} \mid \mathtt{v} = \mathtt{m.get(k)} \mid \mathtt{m.remove(k)} \mid \mathtt{v} = \mathtt{null} \\
&\mid \mathtt{v} = \mathtt{m.putIfAbsent(k, v)} \mid \mathtt{v} = \mathtt{new\ Value()} \mid \mathtt{assert(b)} \\
\mathtt{b} ::= &\ \mathtt{x} == \mathtt{null} \mid \mathtt{x}! = \mathtt{null} \mid \mathtt{m.containsK(k)} \mid !\mathtt{m.containsK(k)}
\end{aligned}
$$

The above fragment captures some representative operations from the Java 7 class `java.util.concurrent.ConcurrentMap`.[5] We represent by `m` the map shared among all the transactions, and `k` a 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 Java library semantics, our language supports (i) $\mathtt{v} = \mathtt{m.get(k)}$ 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) $\mathtt{v} = \mathtt{m.putIfAbsent(k, v)}$ relates `k` to `v` in `m` if `k` is already in `m` and returns the previous value it was related to, (iv) $\mathtt{v} = \mathtt{new\ Value(...)}$ creates a new value, and (v) $\mathtt{v} = \mathtt{null}$ assigns `null` to variable `v`. In addition, our language supports a standard `assert(b)` statement that lets the execution continue iff the given Boolean condition holds. In particular, the language supports checking whether a variable is `null`, and if the map contains a key. This is necessary to support conditional and loop statements.

Concrete Domain and Semantics. We begin by instantiating the state of a transaction t to the language above. Let Var and Ref be the sets of variables and references,

[5] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentMap.html

$$
\begin{aligned}
&\mathbb{C}[\![\texttt{m.put(k,v)}, (e,m)]\!] = (e, m[e(\texttt{k}) \mapsto e(\texttt{v})])\\
&\mathbb{C}[\![\texttt{v} = \texttt{m.get(k)}, (e,m)]\!] = (e[\texttt{v} \mapsto m(e(\texttt{k}))], m)\\
&\mathbb{C}[\![\texttt{m.remove(k)}, (e,m)]\!] = (e, m[e(\texttt{k}) \mapsto \texttt{null}])\\
&\mathbb{C}[\![\texttt{v} = \texttt{m.putIfAbsent(k,v)}, (e,m)]\!] = (e[\texttt{v} \mapsto m(n)], m') :\\
&\qquad\qquad\qquad m' = \begin{cases} m[n \mapsto e(\texttt{v})] & \text{if } m(e(\texttt{k})) = \texttt{null}\\ m & \text{otherwise}\end{cases}\\
&\mathbb{C}[\![\texttt{v} = \texttt{new Value()}, (e,m)]\!] = (e[v \mapsto \mathsf{fresh}(\texttt{t})], m)\\
&\mathbb{C}[\![\texttt{v} = \texttt{null}, (e,m)]\!] = (e[v \mapsto \{\texttt{null}\}], m)\\
&\mathbb{C}[\![\texttt{assert(x == null)}, (e,m)]\!] = (e,m) \text{ if } e(\texttt{x}) = \texttt{null}\\
&\mathbb{C}[\![\texttt{assert(x! = null)}, (e,m)]\!] = (e,m) \text{ if } e(\texttt{x}) \neq \texttt{null}\\
&\mathbb{C}[\![\texttt{assert(m.containsK(k))}, (e,m)]\!]\\
&\quad = (e,m) \text{ if } m(e(\texttt{k})) \neq \texttt{null}\\
&\mathbb{C}[\![\texttt{assert(!m.containsK(k))}, (e,m)]\!]\\
&\quad = (e,m) \text{ if } m(e(\texttt{k})) = \texttt{null}
\end{aligned}
$$

Fig. 3: Concrete semantics

respectively. Keys and values are identified by concrete references, and we assume `null` is in Ref. We define by $\mathsf{Env} : \mathsf{Var} \rightarrow \mathsf{Ref}$ the environments relating local variables to references. A map is then represented as a function $\mathsf{Map} : \mathsf{Ref} \rightarrow \mathsf{Ref}$, relating keys to values. The value `null` represents that the related key is not in the map. A single concrete state is a pair made by an environment and a map. Formally, $\Sigma = \mathsf{Env} \times \mathsf{Map}$. As usual in abstract interpretation, we collect a set of states per program point. Therefore, our concrete domain is made by elements in $\wp(\Sigma)$, and the lattice relies on standard set operators. Formally, $\langle \wp(\Sigma), \subseteq, \cup \rangle$.The concrete semantics are given to the right. Figure 3 defines the concrete semantics. For the most part, it formalizes the API specification of the corresponding Java method. Note that `assert` is defined only on the states that satisfy the given Boolean conditions. n represents a fresh concrete node in the semantics of `putIfAbsent`. In this way, the concrete semantics filters out only the states that might execute a branch of an `if` or `while` statement.

Abstract Domain. Let $\mathsf{HeapNode}$ be the set of abstract heap nodes with $\texttt{null} \in \mathsf{HeapNode}$. Both keys and values are abstracted as heap nodes. As usual with heap abstractions, each heap node might represent one or many concrete references. Therefore, we suppose that a function $\mathsf{isSummary} : \mathsf{HeapNode} \rightarrow \{\mathsf{true}, \mathsf{false}\}$ is provided; $\mathsf{isSummary}(n)$ returns true if n might represent 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 is not in the map. For instance, $[n_1 \mapsto \{\texttt{null}, n_2\}]$ 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. Abstract state $([\mathtt{name} \mapsto \{n_1\}], [n_1 \mapsto \{\mathtt{null}\}])$ represents that key `name` is not in the map, while $([\mathtt{name} \mapsto \{n_1\}], [n_1 \mapsto \{n_2\}])$ represents that it is in the map, and it is related to some value n_2. Finally, $([\mathtt{name} \mapsto \{n_1\}], [n_1 \mapsto \{\mathtt{null}, n_2\}])$ represents that `name` (i) might not be in the map, *or* (ii) is in the map related to value n_2.

Concretization function. We define the concretization function $\gamma_\Sigma : \Sigma \rightarrow \wp(\Sigma)$ that, given an abstract state, returns the set of concrete states it represents. First of all, we assume that a function concretizing abstract heap nodes to concrete references is given. Formally, $\gamma_{\mathsf{Ref}} : \mathsf{HeapNode} \rightarrow \wp(\mathsf{Ref})$. We assume that this concretization function concretizes `null` into itself ($\gamma_{\mathsf{Ref}}(\mathtt{null}) = \{\mathtt{null}\}$), and that it is coherent w.r.t. the information provided by isSummary ($\neg\mathsf{isSummary}(n) \Leftrightarrow |\gamma_{\mathsf{Ref}}(n)| = 1$).

The concretization of abstract environments relates each variable in the environment to a reference concretized from the node it is in relation with. Similarly, the concretization of abstract maps relates a reference concretized from a heap node representing a key with a reference concretized from a node representing a value. Finally, the concretization of abstract states applies point-wisely the concretization of environments and maps, formalized as:

$$\gamma_{\mathsf{Env}}(e) = \{\lambda x.r : x \in dom(e) \wedge \exists n \in e(x) : r \in \gamma_{\mathsf{Ref}}(n)\} \qquad \gamma_\Sigma(\bot) = \emptyset$$
$$\gamma_{\mathsf{Map}}(m) = \{\lambda r_1.r_2 : \exists n_1 \in dom(m) : r_1 \in \gamma_{\mathsf{Ref}}(n_1) \wedge \exists n_2 \in m(n_1) : r_2 \in \gamma_{\mathsf{Ref}}(n_2)\}$$
$$\gamma_\Sigma(e, m) = \{(e', m') : e' \in \gamma_{\mathsf{Env}}(e) \wedge m' \in \gamma_{\mathsf{Map}}(m)\}$$

Lemma 2 (Soundness of the domain). *The abstract domain is a sound approximation of the concrete domain, that is, they form a Galois connection [3]. Formally,* $\langle \wp(\Sigma), \subseteq, \cup \rangle \underset{\alpha_\Sigma}{\overset{\gamma_\Sigma}{\leftrightarrows}} \langle \Sigma, \dot{\subseteq}, \dot{\cup} \rangle$ *where* $\alpha_\Sigma = \lambda \mathsf{X}. \cap \{\mathsf{Y} : \mathsf{Y} \subseteq \gamma_\Sigma(X)\}$.

Proof Sketch. γ_Σ *is a complete meet-morphism since it produces all possible environments and maps starting from a given reference concretization. Then,* α_Σ *is well-defined since* γ_Σ *is a complete* $\cap$*-morphism. The fact that it forms a Galois connection follows immediately from the definition of* α_Σ *(Proposition 7 of [4]).*

Running example. Consider again abstract state $\sigma = ([\mathtt{name} \mapsto \{n_1\}], [n_1 \mapsto \{\mathtt{null}, n_2\}])$. Suppose γ_{Ref} concretizes n_1 and n_2 into $\{\#1\}$ and $\{\#2\}$, respectively. Then σ is concretized into states $([\mathtt{name} \mapsto \#1], [\#1 \mapsto \mathtt{null}])$ representing that `name` is not in the map and $([\mathtt{name} \mapsto \#1], [\#1 \mapsto \#2])$ representing that `name` is in the map is related to the value pointed-to by reference $\#2$.

Abstract Semantics. Figure 4 is the abstract semantics of statements and Boolean conditions, that, given an abstract state and a statement or Boolean condition of the language introduced above, returns the abstract state resulting from the evaluation of the given statement on the given abstract state. As usual in abstract interpretation-based static analysis [3], this operational abstract semantics is the basis for computing a fixpoint over a CFG representing loops and conditional statements. We focus the formalization on

$$\mathbb{S}[\![\mathtt{v} = \mathtt{new\ Value}(), (e,m)]\!] = (e[v \mapsto \mathsf{fresh}(\mathtt{t})], m) \qquad (\mathtt{new})$$
$$\mathbb{S}[\![\mathtt{v} = \mathtt{null}, (e,m)]\!] = (e[v \mapsto \{\mathtt{null}\}], m) \qquad (\mathtt{nlas})$$
$$\mathbb{S}[\![\mathtt{v} = \mathtt{m.get(k)}, (e,m)]\!] = (e[\mathtt{v} \mapsto \textstyle\bigcup_{n \in e(k)} m(n)], m) \qquad (\mathtt{get})$$
$$\mathbb{S}[\![\mathtt{m.put(k,v)}, (e,m)]\!] \qquad (\mathtt{put})$$
$$= \begin{cases} (e, m[n \mapsto e(\mathtt{v})]) & \text{if } e(\mathtt{k}) = \{n\} \wedge \neg\mathsf{isSummary}(n) \\ (e, m[n \mapsto m(n) \cup e(\mathtt{v}) : n \in e(\mathtt{k})]) & \text{otherwise} \end{cases}$$
$$\mathbb{S}[\![\mathtt{m.remove(k)}, (e,m)]\!] \qquad (\mathtt{rmv})$$
$$= \begin{cases} (e, m[n \mapsto \{\mathtt{null}\}]) & \text{if } e(\mathtt{k}) = \{n\} \wedge \neg\mathsf{isSummary}(n) \\ (e, m[n \mapsto m(n) \cup \{\mathtt{null}\} : n \in e(\mathtt{k})]) & \text{otherwise} \end{cases}$$
$$\mathbb{S}[\![\mathtt{v} = \mathtt{m.putIfAbsent(k,v)}, (e,m)]\!] \qquad (\mathtt{pIA})$$
$$= (\pi_1(\mathbb{S}[\![\mathtt{v} = \mathtt{m.get(k)}, (e,m)]\!]), m') :$$
$$m' = \begin{cases} m[n \mapsto e(\mathtt{v})] & \text{if } e(\mathtt{k}) = \{\mathtt{n}\} \wedge m(n) = \{\mathtt{null}\} \\ m[n \mapsto m(n) \cup e(\mathtt{v}) : n \in e(\mathtt{k})] & \text{if } \mathtt{null} \in m(n) \wedge |m(n)| > 1 \\ m & \text{otherwise} \end{cases}$$
$$\mathbb{S}[\![\mathtt{assert(x == null)}, (e,m)]\!] \qquad (\mathtt{null})$$
$$= \begin{cases} (e[\mathtt{x} \mapsto \{\mathtt{null}\}], m) & \text{if } \mathtt{null} \in e(\mathtt{x}) \\ \bot & \text{otherwise} \end{cases}$$
$$\mathbb{S}[\![\mathtt{assert(x! = null)}, (e,m)]\!] \qquad (!\mathtt{null})$$
$$= \begin{cases} (e[\mathtt{x} \mapsto e(\mathtt{x}) \setminus \{\mathtt{null}\}], m) & \text{if } \exists n \in \mathsf{HeapNode} : n \neq \mathtt{null} \wedge n \in e(\mathtt{x}) \\ \bot & \text{otherwise} \end{cases}$$
$$\mathbb{S}[\![\mathtt{assert(m.containsK(k))}, (e,m)]\!] \qquad (\mathtt{cntK})$$
$$= \begin{cases} \bot & \text{if } \forall n \in e(\mathtt{k}) : m(n) = \{\mathtt{null}\} \\ (e, m[n \mapsto m(n) \setminus \{\mathtt{null}\}]) & \text{if } e(\mathtt{k}) = \{n\} \wedge \neg\mathsf{isSummary}(n) \wedge m(n) \neq \{\mathtt{null}\} \\ (e,m) & \text{otherwise} \end{cases}$$
$$\mathbb{S}[\![\mathtt{assert(!m.containsK(k))}, (e,m)]\!] \qquad (!\mathtt{cntK})$$
$$= \begin{cases} \bot & \text{if } \forall n \in e(\mathtt{k}) : \mathtt{null} \notin m(n) \\ (e, m[n \mapsto \{\mathtt{null}\}) & \text{if } e(\mathtt{k}) = \{n\} \wedge \neg\mathsf{isSummary}(n) \wedge \mathtt{null} \in m(n) \\ (e,m) & \text{otherwise} \end{cases}$$

Fig. 4: Formal definition of the abstract semantics.

abstract states in $\mathsf{Env} \times \mathsf{Map}$, since in case of $\bot$ the abstract semantics always returns $\bot$ itself.

(`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, (`nlas`) relates the given variable to the singleton $\{\mathtt{null}\}$. (`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. (`put`) relates `k` to `v` in the map. In particular, if `k` points to a unique non-summary 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. Similarly to (`put`), (`rmv`) removes `k` from the map (by relating it to the singleton $\{\texttt{null}\}$) iff `k` points to a unique concrete node. Otherwise, it conservatively adds the heap node `null` to the heap nodes related to all the values pointed by `k`. $(\texttt{pIA})$ updates the map like $(\texttt{put})$ but only if the updated key node might have been absent, that is, when $\texttt{null} \in m(n)$. The abstract semantics on Boolean conditions produces $\perp$ statements if the given Boolean condition cannot hold on the given abstract semantics. Therefore, $(\texttt{null})$ returns $\perp$ if the given variable `x` cannot be `null`, or a state relating `x` to the singleton $\{\texttt{null}\}$ otherwise. Vice-versa, $(!\texttt{null})$ returns $\perp$ if `x` can be only null, or a state relating `x` to all its previous values except `null` otherwise. Similarly, $(\texttt{cntK})$ returns $\perp$ 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, $(!\texttt{cntK})$ returns $\perp$ if `k` is surely in the map.

Lemma 3 (Soundness of the semantics). *The abstract semantics is a sound approximation of the concrete semantics. Formally,* $\forall \texttt{st}, (e, m) \in \boldsymbol{\Sigma} : \gamma_{\boldsymbol{\Sigma}}(\mathbb{S}[\![\texttt{st}, (e, m)]\!]) \supseteq \mathbb{C}[\![\texttt{st}, \gamma_{\boldsymbol{\Sigma}}(e, m)]\!]$*, where* $\mathbb{C}$ *represents the pointwise application of the concrete semantics to a set of concrete states.*

Proof Sketch. Follows from case splitting on the statement, and by definition of the concrete and abstract semantics.

Running example. Consider again the code of method getConvertor, and suppose that the Boolean flag `create` is `true`. When we start from the abstract state $([\texttt{name} \mapsto \{n_1\}], [n_1 \mapsto \{\texttt{null}\}])$ (representing that `name` is not in the map), we obtain the abstract state $\sigma = ([\texttt{name} \mapsto \{n_1\}, \texttt{conv} \mapsto \{\texttt{null}\}], [n_1 \mapsto \{\texttt{null}\}])$ after the first statement by rule $(\texttt{get})$. Then the semantics of the Boolean condition of the `if` statements at line 3 applies rule $(\texttt{null})$ (that does not modify the abstract state) since `conv` is `null`, and we assumed `create` is `true`. Lines 4 and 5 applies rules $(\texttt{new})$ and $(\texttt{pIA})$, respectively. Supposing that $\mathsf{fresh}(t)$ returns n_2, we obtain $\sigma' = ([\texttt{name} \mapsto \{n_1\}, \texttt{conv} \mapsto \{n_2\}], [n_1 \mapsto n_2])$. We then join this state with the one obtained by applying rule $(!\texttt{null})$ to σ (that is, $\perp$) obtaining σ' itself. The result of this example represents that, when you start the computation passing a key `name` that is not in the map and `true` for the Boolean flag `create`, after executing method `getConvertor` in isolation you obtain a map relating `name` to the new object instantiated at line 4.

5 Inferring Corrective Targets

We now apply the abstract semantics $\mathbb{S}$ to infer corrective targets. For this paper, we support a restricted transactional model. In particular, we assume that there are n transactions that start the execution together, each transaction commits only once, and all the transactions commit together at the end of the execution. With these assumptions, we can define a system that perform a *global* corrective synchronization at the end of the execution. We leave more expressive inference for future work.

Serialized CFG. We apply the abstract semantics defined in Section 4 to compute suitable corrective targets. In particular, we need that these targets are reachable from the same *entry state* through a *serializable execution*. Therefore, we build a CFG that represents certain specific *serialized* executions. In particular, we assume that we have k distinct types of transactions, and we build up a serialized CFG that represents a serialized execution of *at least* 2 instances of each type of transaction.

Let $\{c^1, ..., c^k\}$ be the code of k different transactions. For each transaction type i, we create three static transaction identifiers t^i_1, t^i_2, and t^i_n. t^i_1 and t^i_2 represent precisely two concrete instances of c^i, while t^i_n is a *summary* abstract instance representing many concrete instances of c^i. We then build a CFG representing a serialized execution of all these abstract transactions. In particular, each transaction type c^i leads to a CFG tc^i that executes (i) first t^i_1, (ii) then t^i_n inside a non-deterministic loop (to over-approximate many instances of c^i)), and (iii) finally t^i_2. While the choice of having the two concrete transaction instances before and after the summary instance is arbitrary and other solutions are possible, we found this solution particularly effective in practice as we will show in Section 6. The overall serialized CFG tc is then built by concatenating the CFGs of all these transactions.

Let $\mathcal{T}^\#$ be the set of abstract transactions, that is $\mathcal{T}^\# = \{t^i_j : i \in [1..k], j \in \{1, 2, n\}\}$. Then our semantics on a serialized CFG returns a function in $\Phi : \mathcal{T}^\# \to \Sigma$.
Running example. Similarly to other synchronization approaches, such as transactional boosting [11] and foresight [7], applying corrective synchronization at the data-structure level requires a commutativity specification. In the case of concurrent maps, a simple and effective specification is in terms of the accessed key. Hence, for the getConvertor code, we build a serialized CFG where all the transactions share the same key `name` (to induce potential conflicts). For the sake of presentation, we set `create` to `true`. The serialized CFG consists of the sequence of transactions $t_1; t^\star_n; t_2$, where $t^\star_n$ represents that the code of t_n is inside a loop.

Suppose now we analyze this serialized CFG starting from the abstract state $([\mathtt{name} \mapsto \{n_1\}], [n_1 \mapsto \{\mathtt{null}\}])$. The abstract semantics computes the following abstract post-state: $([\mathtt{name} \mapsto \{n_1\}, \mathtt{conv}_1 \mapsto \{n^1_1\}, \mathtt{conv}_n \mapsto \{n^1_1\}, \mathtt{conv}_2 \mapsto \{n^1_1\}], [n_1 \mapsto \{n^1_1\}])$ (where n^a_b represents the a-th node instantiated by transaction $\mathtt{t_b}$, and $\mathtt{conv}_c$ represents the local variable `conv` of transaction $\mathtt{t_c}$). Intuitively, this result means that, if we run a sequence of transactions executing the code of method `getConvertor` with a map that does not contain key `name`, then at the end of the execution of all transactions we will obtain a map relating `name` to the value generated by the first transactions, and all the transactions will return this value.

Extracting Possible Corrective Targets. First notice that, given a transaction t, the corr rule of the transition system introduced in Section 3 requires that the state the system corrects to is reachable starting from the state at the beginning of the execution of t (retrieved by $\Gamma(t) = s$ [n.b. abuse of notation]) producing the same shared log (formally, $s \rightsquigarrow (T, \mu', \sigma, L)$). Since in our specific instance of the transition system we suppose that all the transactions start together, we assume that there is a unique entry state σ_0 (formally, $\forall t \in T : \Gamma(t) = \sigma_0$). In addition, since all the transactions commit together at the end, we have complete control over the shared log, and when we correct the shared log is always empty, and the shared state is identical to the initial shared state. Therefore, given these restrictions, we only need to compute a μ' such that $\Gamma(t) \rightsquigarrow (T, \mu', \sigma_0, \epsilon)$.

We compute possible corrective targets on the serialized CFG tc using the abstract semantics $\mathbb{S}$. In particular, we need to compute corrective targets that, given an entry state representing a precise observable entry state, are reachable through a serialized execution. However, an abstract state in Σ might represent multiple concrete states. For instance $([\mathtt{k} \mapsto \{n_1\}], [n_1 \mapsto \{\mathtt{null}, n_2\}])$ 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, and it cannot be used to define a corrective target. Therefore, we define a predicate single : $\Sigma \rightarrow \{\mathsf{true}, \mathsf{false}\}$ that, given an abstract state, holds iff it represents an exact concrete state. Formally,

$$\mathsf{single}(e, m) \Leftrightarrow \bigwedge \begin{cases} \forall \mathtt{x} \in \mathit{dom}(e) \colon |e(\mathtt{x})| = 1 \wedge e(\mathtt{x}) = \{n_1\} \wedge \neg \mathsf{isSummary}(n_1) \\ \forall n \in \mathit{dom}(m) : |m(n)| = 1 \wedge m(n) = \{n_2\} \wedge \neg \mathsf{isSummary}(n_2) \end{cases}$$

Note that in general the concretization of an abstract state is not computable. Therefore, we rely on single to check if an abstract state represents one precise concrete state.

Lemma 4. $\forall (e, m) \in \Sigma : \mathsf{single}(e, m) \Rightarrow |\gamma_{\Sigma}(e, m)| = 1$

Proof Sketch (Proof Sketch). By definition of single*,* $\neg\mathsf{isSummary}(n)$ *for all the nodes* n *in* e *or* n*. By definition of* isSummary *we have that* $|\gamma_{\mathsf{Ref}}(n)| = 1$*. Thanks to this result, combined with the definition of* γ_{Σ}*, we obtain that* $|\gamma_{\Sigma}(e, m)| = 1$.

The definition of single is extended to states $\phi \in \Phi$ by checking if single holds for all the local states in ϕ. We build up a set of possible entry states $S \subseteq \Phi$ such that $\forall \phi \in S : \mathsf{single}(\phi)$, and we compute the exit states on the serialized CFG tc for all the possible entry states, filtering out only the ones that represents an exact concrete state. Note that since we have a finite number of abstract transactions, and each transaction has a finite number of parameters, we can build up a finite set of entry states representing all the possible concrete situations. Note that, while in general an abstract state rarely represents a precise single concrete state, this is the case for most of the cases we dealt with as shown by our experimental results. This happens since our static analysis targets a specific data structure (maps), and tracks very precise symbolic information on it.

We then use the results of the abstract semantics $\mathbb{S}$ to build up a function corrTarg that relates each entry state to a set of possible exit states: $\mathsf{corrTarg}(\mathsf{T}, \mathsf{S}) = [\phi \mapsto \{\phi' : \phi' \in \mathbb{S}[\![tc, \phi]\!] \wedge \mathsf{single}(\phi')\} : \phi \in S]$.

Running example. Starting from the entry state $([\mathtt{name} \mapsto \{n_1\}], [n_1 \mapsto \{\mathtt{null}\}])$, the exit state computed by the abstract semantics is $([\mathtt{name} \mapsto \{n_1\}, \mathtt{conv}_1 \mapsto \{n_1^1\}, \mathtt{conv}_n \mapsto \{n_1^1\}, \mathtt{conv}_2 \mapsto \{n_1^1\}], [n_1 \mapsto \{n_1^1\}])$. This state satisfies the predicate single since it represents a precise concrete state. Therefore, the relation between this entry and exit state is part of corrTarg.

Dynamic Corrective Synchronization. In our model, when we start the execution we have a finite number of concrete instances of each type of transaction. We denote by $\mathcal{T} = \{s_j^i : j \in [0..m] \wedge i \in [1..k_j]\}$ the set of identifiers of concrete transactions, where m is the number of different types of transactions, k_j is the number of instances of transaction j, and s_j^i represents the j-th instance of the i-th type of transaction.

We can then bind abstract transaction identifiers to concrete ones. Since the set of abstract transactions is defined as $\mathcal{T}^{\#} = \{t_j^i : i \in [1..k], j \in \{1, 2, n\}\}$, we bind the first two concrete identifiers to the corresponding abstract identifiers, and all the others to the n abstract instance. We formally define the concretization of transaction identifiers as

follows: $\gamma_{\mathcal{T}}(T) = [t_j^i \mapsto \{s_j^{i'} : (i \in \{1,2\} \Rightarrow i' = i) \vee 3 \leq i' \leq k_j\} : t_j^i \in T]$. We can now formalize the concretization of abstract states in Φ by relying on the concretization of local states and transaction identifiers. $\gamma_{\Phi}(\phi) = \{t \mapsto \sigma : \exists t' \in dom(\phi) : t \in \gamma_{\mathcal{T}}(t) \wedge \sigma \in \gamma_{\Sigma}(\phi(t))\}$.
We now prove that targets computed by corrTarg satisfy the premise of corr.

Theorem 3. *Let* $t = \mathsf{corrTarg}(\mathsf{T}, \mathsf{S})$ *be the results computed by our system. Then* $\forall \sigma_0 \in \gamma_{\Phi}(\phi_0), \sigma_n \in \gamma_{\Phi}(\phi_n) : \phi_0 \in dom(t) \wedge \phi_n \in t(\phi_0)$ *we have that* $\sigma_0 \rightsquigarrow \sigma_n$.

Proof Sketch. By definition of corrTarg, we have that both $\mathsf{single}(\phi_0)$ and $\mathsf{single}(\phi_n)$ hold. Therefore, by Lemma 4 we have that $\gamma_{\Sigma}(\phi_0) = \{\sigma_0\}$ and $\gamma_{\Sigma}(\phi_n) = \{\sigma_n\}$. In addition, by definition of corrTarg we have that $\phi_n \in \mathbb{S}[\![tc, \phi_0]\!]$. Then, by lemma 3 (soundness of the abstract semantics) we have that σ_n is exactly what is computed by the concrete semantics on the given program starting from σ_0, that is, $\sigma_0 \rightsquigarrow \sigma_n$.

Discussion. corrTarg returns a set of possible exit states given an entry state. This means that, given a concrete incorrect post-state, we can choose the exit state produced by corrTarg that requires a *minimal* correction to the incorrect post state. In this way, we would minimize the runtime overhead of adjusting the concrete state. The target state can be chosen by calculating the number of operations we need to apply to correct the post-state, and select the one with the minimal number. This might be further optimized by hashing the correct post states computed by corrTarg based on similarity. However, we did not investigate this aspect since in our experiments the overhead of correcting the post-state was already almost negligible by choosing a random target. We believe that this is due to our specific setting, that is, concurrent maps. In fact, in this scenario the corrective operations that we have to apply are to put or remove an element, and the corrections always required very few of them. We believe that other data structures (e.g., involving ordering of elements like lists) might require more complicated corrections, and we plan to investigate them as future work.

6 Preliminary Implementation

We now report encouraging results of our preliminary implementation. We have created a Java implementation of our static analysis for composed Map operations (see Section 5). Given n types of transactions, our implementation builds a serialized CFG (Section 5) and then computes a fixpoint over it relying on the abstract semantics (Section 4). We support all the operations listed in Section 4. Static analysis running times are negligible compared to the rest of the process, and the analysis converges always in less than a second. Therefore, we do not report the running times of the static analysis.

As explained earlier, the interface with the runtime system is a relational corrective specification mapping pre-states to sets of post-states that are obtainable via serializable execution of the transactions from the pre-state. As a partial example, $[\mathsf{k} \mapsto \bot, \mathsf{v} \mapsto v] \rightsquigarrow \{[\mathsf{k} \mapsto v, \mathsf{v} \mapsto v]\}$ denotes that if we started from a pre-state where k was not in the map and the value passed to the function was v, then in the post-state the key k is made to point to the value v pointed-to by the second argument v in the pre-state. The runtime system S is parameterized by the specification, which it loads at the beginning of the concurrent run. As discussed in Section 5, in our current prototype all transactions are assumed to start simultaneously. This scenario is useful, for example, in loop parallelization. Each concrete transaction is mapped to its abstract counterpart. The mapping process also binds the concrete arguments of the transaction (i.e., the concrete object references) to their symbolic counterparts (e.g., the k and v symbols above).

During execution, the runtime system monitors commit events. In our prototype, we limit transactions to a single commit point before completion. Corrective synchronization occurs on failed commits, in which case the transaction's shared log, local state and return value are all (potentially) modified according to the corrective specification.

Summary of Subjects and Experiments. We conducted experiments running our implementation on four subjects, all of which are taken from popular open-source code bases and have been used in past studies[23,24]. We considered workload size and concurrency level, ranging from 2 to 23 threads, and summarizing over 10 runs. In each case,we compared against (i) a pessimistic concrete-level variant of STM, as available via version 1.3 of the Deuce STM (the latest version)[6], and (ii) a lock-based synchronization algorithm boosted with Map semantics [11], such that the locks are of the same grain as their corresponding abstract locks in boosted STM. We ran our experiments on two Intel Xeon 2.90GHz (16 cores) CPUs with 132GB of RAM.

For lack of space, detailed experimental results, considering factors such as number of threads and size of the workload, are omitted. The table on the right reports the relative gain of STM and corrective synchronization w.r.t. lock-based synchronization. We aggregate performance results, averaging across all workloads and concurrency levels. These results show that, on average, corrective synchronization leads to a gain that is twice the one obtained by STM.

	Pessimistic		Corrective	
	wload.	conc.	wload.	conc.
Tomcat	1.5%	3%	29%	30%
dyuproject	18%	24%	30%	29%
Flexive	17%	14%	29%	30%
Gridkit	20%	16%	32%	31%
average	**14%**	**14%**	**30%**	**30%**

For completeness, we also note the absolute running times, as min/max intervals in seconds, for the lock-based solution for the workload and concurrency configurations respectively: Tomcat – [6,10], [7,12]; dyuproject – [6,9], [7,11]; Flexive – [6,10], [7,11]; and Gridkit – [6,9], [7,11]. The numbers are encouraging, indicating improvement over both locks and STM. More careful engineering, beyond our current prototype implementation, is likely to make the improvement more significant.

7 Related work

To our knowledge, existing solutions to the problem of correct synchronization assume either the ability to prevent bad interleavings or the ability to roll back execution. We focus our survey of related research on solutions for optimizing the rollback mechanism, and also discuss works on synchronization synthesis backed by static program analysis and on merging state mutations by concurrent threads.

There are two main optimizations to *decrease rollback overhead*: reducing either abort rate or the extent to which a conflicted transaction rolls back. Different solutions

[6] `https://github.com/DeuceSTM/DeuceSTM`

have been proposed in each direction [11,17,26]. Others leverage available nondeterminism [25]. None of these approaches perform corrective synchronization. A well-known solution to restrict the extent to which a transaction rolls back is checkpointing [14,5] or nested transactions [20,1]. Elastic transactions [6] avoid wasted work by splitting into multiple pieces. The Push/Pull model [15] also uses local/shared logs, and is flexible enough to express rollback-based transactions but nor corrective synchronization.

In our solution, static analysis is used to identify admissible shared-state configurations to correct to from a given input state. Multiple past works on synchronization synthesis have also *relied on static analysis*, albeit for the extraction of other types of information. Golan-Gueta et al. [8] utilize static analysis to compute a conservative approximation of the possible actions that a transaction may perform in its future from a given intermediate point. This still pessimistic approach enables more granular synchronization compared to the worst-case assumption that the transaction may perform any action in its future. Autolocker [19] applies static analysis to determine a correct locking policy. Hawkins et al. [9] ensure correct synchronization by construction. Prountzos et al. [21] optimize the Galois system [17] via static shape analysis [22].

Finally, we note solutions based on merging, or combining, the effects of concurrent threads. Burckhardt and Leijen propose *concurrent revisions* [2], inspirated by version control systems. The idea is to specify a (custom) merge function, based on a revision calculus, such that concurrent state mutations can be reconciled in a deterministic manner. Somewhat similarly, Hendler *et al.* introduce *flat combining* [10]. The idea is to synchronize concurrent accesses to a shared data structure D by having threads post their updates to D into a common list as thead-local records, where a single thread at a time acquires the lock on D, combines and applies the updates, and writes the results back to the threads' request fields. Contrary to these two paradigms, our approach builds on thread-level state correction. This bypasses the need for a coarse-grained lock, as in flat combining, and — unlike concurrent revisions — ensures serializability.

8 Conclusion and Future Work

We have presented an alternative to the lock- and retry-based synchronization methods that we dub *corrective synchronization*. The key insight is to correct a bad execution, rather than aborting/retrying the transaction or conservatively avoiding the bad execution in the first place. We have explored an instantiation of corrective synchronization for composed operations over ConcurrentMaps, where correct states are computed via abstract interpretation. Experimental results with a prototype implementation are encouraging.

There are several directions for future work. First, one may explore other variants of corrective synchronization. For example, one may want to allow multiple threads to decide together corrective target states. One could also improve the abstract domain beyond maps, perhaps using existing domains. Another direction is to integrate corrective synchronization into larger-scale software systems and perform a deep experimental evaluation over the spectrum of synchronization techniques. As part of such an effort, one could develop compositional synchronization methods that integrate corrective synchronization with lock- and STM-based synchronization.

References

1. C. Beeri, P. A. Bernstein, N. Goodman, M. Y. Lai, and D. E. Shasha. A concurrency control theory for nested transactions (preliminary report). In *Proceedings of the 2nd annual ACM symposium on Principles of distributed computing (PODC'83)*, pages 45–62, New York, NY, USA, 1983. ACM Press.
2. S. Burckhardt and D. Leijen. Semantics of concurrent revisions. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 116–135, 2011.
3. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL '77*. ACM Press, 1977.
4. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.
5. Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *J. Supercomput.*, 65:1302–1326, 2013.
6. Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings*, pages 93–107, 2009.
7. G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Concurrent libraries with foresight. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 263–274, 2013.
8. Guy Golan-Gueta, G. Ramalingam, Mooly Sagiv, and Eran Yahav. Concurrent libraries with foresight. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 263–274, 2013.
9. Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 417–428, 2012.
10. D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, pages 355–364, 2010.
11. M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 207–216, 2008.
12. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture. San Diego, CA, May 1993*, pages 289–300, 1993.
13. Andi Kleen. Scaling existing lock-based applications with lock elision. *Commun. ACM*, 57(3):52–56, 2014.
14. Eric Koskinen and Maurice Herlihy. Checkpoints and continuations instead of nested transactions. In Friedhelm Meyer auf der Heide and Nir Shavit, editors, *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*, pages 160–168. ACM, 2008.
15. Eric Koskinen and Matthew J. Parkinson. The push/pull model of transactions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 186–195, 2015.

16. Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.
17. Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 211–222, 2007.
18. A. Matveev and N. Shavit. Towards a fully pessimistic stm model. In *Proc. Workshop on transactional memory (TRANSACT12)*, 2012.
19. Bill McCloskey, Feng Zhou, David Gay, and Eric A. Brewer. Autolocker: synchronization inference for atomic sections. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 346–358, 2006.
20. Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'07)*, pages 68–78, New York, NY, USA, 2007. ACM Press.
21. Dimitrios Prountzos, Roman Manevich, Keshav Pingali, and Kathryn S. McKinley. A shape analysis for optimizing parallel graph programs. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 159–172, 2011.
22. Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24:217–298, 2002.
23. Ohad Shacham, Nathan Grasso Bronson, Alex Aiken, Mooly Sagiv, Martin T. Vechev, and Eran Yahav. Testing atomicity of composed concurrent operations. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 51–64, 2011.
24. Ohad Shacham, Eran Yahav, Guy Golan-Gueta, Alex Aiken, Nathan Grasso Bronson, Mooly Sagiv, and Martin T. Vechev. Verifying atomicity via data independence. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 26–36, 2014.
25. Omer Tripp, Eric Koskinen, and Mooly Sagiv. Turning nondeterminism into parallelism. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 589–604, 2013.
26. Omer Tripp, Greta Yorsh, John Field, and Mooly Sagiv. Hawkeye: Effective discovery of dataflow impediments to parallelization. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 207–224, New York, NY, USA, 2011. ACM.