Corrective Synchronization

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

[1] JuliaSoft
[2] Google
[3] IBM Research
[4] Yale University

Abstract. Concurrency control is a challenging problem. While some thread interleavings are admissible, there are certain interleaving scenarios that lead to inadmissible program states. Broadly speaking, there are two main paradigms for avoiding such situations: *pessimistic synchronization* reduces parallelism to mitigate interference between threads, while *optimistic synchronization* detects when illegal interleavings have occurred and rolls execution (of one or more threads) back to an admissible point.

We propose a novel synchronization paradigm. In our approach, dubbed *corrective synchronization*, the correctness of multithreaded execution is enforced neither by reducing parallelism nor by rolling back illegal thread interleavings, but by correcting the inadmissible state after the fact. The system automatically compensates, if necessary, for the effects of inadmissible interleavings by modifying the program state as a transaction completes while accounting for the behavior of concurrent transactions. We have proven that corrective synchronization is serializable.

Within the general scope of corrective synchronization, we explore the combination of (i) an offline static analysis to compute correct states w.r.t. the entry states with (ii) a runtime protocol that utilizes the artifacts computed statically to perform online state correction. We instantiate this setup to clients of the Java ConcurrentMap abstract data type (ADT) to ensure safe composition of map operations.

We have created a prototype implementation of this system in Java, which we compare against two mainstream techniques: a lock-based approach boosted with abstract Map semantics as well as a standard software transactional memory (STM). Our experimental evaluation on several real-world benchmarks indicates x2 speedup improvement compared to these techniques. Further, our technique incurs negligible overhead as compared to the original execution.

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 inhibited to ensure serializability [18]. 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:

– *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 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) [10], 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 [11]. 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 [14].

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,[5] where a shared Map object, (pointed-to by) _convertors, is manipulated by method getConvertor.[6]

```
1 Convertor getConvertor(String name, boolean create) {
2   Convertor conv = _convertors.get(name);
3   if (conv==null && create) {
4     conv = new Convertor(name);
5     _convertors.putIfAbsent(name, conv); }
6   return conv; }
```

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

Assume that different threads invoking this method are all attempting to simultaneously obtain the Convertor object related to the same key, which is not yet part of the map. Doing so optimistically would lead to multiple rollbacks (even under boosted conflict detection [9], since the operations due to different threads do not commute), 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 new Convertor() is an expensive operation.

Corrective synchronization. In this paper, we take a first step in formulating and exploring a novel synchronization paradigm, which is conceptually different from both the

[5] https://code.google.com/archive/p/dyuproject/

[6] The code in Figure 1 is a slight variation of the original code of dyuproject where we simplified the syntax for the sake of clarity

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:

$$
\begin{aligned}
[\quad & T_1\text{: get}()/\text{null} \rightarrow T_2\text{: get}()/\text{null} \\
\hookrightarrow\ & T_1\text{: if}(\ldots) \rightarrow T_2\text{: if}(\ldots) \\
\hookrightarrow\ & T_1\text{: new } \ldots/o_1 \rightarrow T_2\text{: new } \ldots/o_2 \\
\hookrightarrow\ & T_1\text{: putIfAbsent}()/\text{null} \rightarrow T_2\text{: putIfAbsent}()/o_1 \\
\hookrightarrow\ & T_1\text{: return } o_1 \rightarrow T_2\text{: return } o_2 \quad]
\end{aligned}
$$

This history is clearly nonserializable. 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 conv to o_1, rather than o_2. Second, we fix the mapping under _convertors for key name 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 conv) for all threads, which would also be the value linked by the key under _convertors. In addition, 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.

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. In particular, we focus on concurrent Java programs whose shared state is encoded as one or more ConcurrentMap instances. The motivation for this initial scope is a series of past studies, which indicate that ConcurrentMap is commonly used in concurrent applications, but often without sufficient synchronization [21,22]. We study several such examples from popular open-source code bases, including Tomcat and Gridkit.

Two important challenges w.r.t. corrective synchronization that we investigate within this scope are (i) how to compute correct poststates; and (ii) given an incorrect poststate, how to efficiently recover the execution to a correct poststate. We tackle these challenges via static analysis based on abstract interpretation. The analysis, equipped with a specialized abstraction for maps, is used to derive the correct poststates in relation to a given prestate.

To gain insight into the merit of corrective synchronization, we have created a prototype implementation for our scope of ConcurrentMap clients. We compare our approach against both STM and lock-based synchronization across different concurrency levels and workload sizes. The results are encouraging: While both STM and corrective synchronization are significantly better than locks, the relative performance improvement thanks to the corrective approach is twice of that achieved by STM.

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. This includes a 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 concurrent maps. We describe the analysis in full formal details.
4. Implementation and evaluation: We have created an implementation of corrective synchronization assuming the shared state is represented as a collection of concurrent maps. We present experimental evidence in favor of corrective synchronization, where our subjects are real-world Java applications.

2 Technical Overview

In this section, we walk the reader through a high-level overview of corrective synchronization. We first describe the conceptual details at a technical level, and then the two main algorithmic steps and how our prototype implementation carries out these steps.

2.1 Conceptual Approach

By corrective synchronization we mean the ability to transform a concurrent run that, in its present state, may not be serializable into a run that is serializable. Stated formally, corrective synchronization is a relationship $h \rightsquigarrow h'$ between histories, such that (i) h and h' share the same initial state, and (ii) h and h' share the same commit log (i.e., they agree on the operations on the shared-state).

The first condition ensures that corrective synchronization yields a feasible outcome. The second is effectively 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 "jump" to another state.

Running example. We illustrate the difference between corrective synchronization and classic optimistic and pessimistic synchronization in Figure 2, where we visually represent concurrent execution of two instances of the code listed in Figure 1 using locks, STM and corrective synchronization (proceeding horizontally left-to-right). Locking serializes execution, and so there is no performance gain whatsoever. As for STM and corrective synchronization, we consider the interleaving scenario specified in Section 1.

Both STM and corrective synchronization, allowing the problematic chain of interleavings, reach a nonserializable state. STM resolves this by retrying the entire

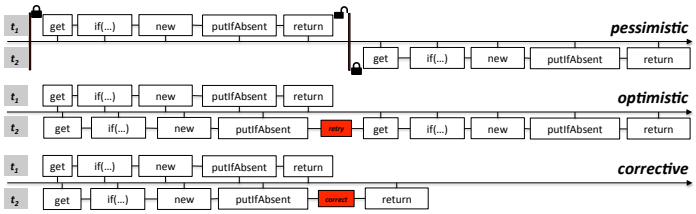

Fig. 2: Interleaved execution of two instances of the method in Figure 1 using lock-, STM- and corrective synchronization

transaction executed by thread t_2. This effectively yields serial execution, similarly to the lock-based 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.

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. Obviously a sound and complete corrective synchronization solution is undecidable. In the following, 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 STM, which the system can default to in the absence of a corrective target.

2.2 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 runtime instrumentation it maps to is minimal compared to tracking the entire execution history. At the same time, however, merely recording initial and current states at runtime does not point back to prefixes h and h'.

Mapping back from pairs of states to prefixes requires an oracle. In our prototype system, the oracle is computed 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 5.1, but note now briefly 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.

Running example. As an illustration of the third point, in our running example (Figure 1) 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.

2.3 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.

In our implementation, the first challenge is addressed 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.

We expand on both of these challenges in Section 6.1. Prior to that, in Sections 4 and 5, we provide a formal statement of corrective synchronization.

2.4 Discussion

3 Transaction Semantics

In this section, we introduce a generic transaction semantics supporting corrective synchronization.

3.1 Notation

In our description of the transition system, we utilize the following semantic domains:

$$
\begin{array}{ll}
c \in \mathcal{C} := \text{command} \quad t \in T \subset \mathcal{T} & := \text{transaction IDs} \\
\sigma \in \Sigma := \text{shared state } \sigma_t & := \text{local state of } t \\
L \in \mathcal{L} := \text{shared log} \quad L_t & := \text{local log of } t \\
s = (T, [t \mapsto (c_t, \sigma_t, L_t)]_{t \in T}, \sigma, L) \in \mathcal{S} & := \text{system state}
\end{array}
$$

Following [13], our transaction semantics uses shared and local logs. 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 and $w(c) \neq \bot$.

3.2 Transition System

We define five transactional events, as follows: The bgn event marks the beginning of a transaction. The cmt event fires when a transaction publishes its outstanding log of operations that affect the shared state to the shared state and log. The end event marks the termination of a transaction. The corr event enables a transaction to modify its local state and log, under certain restrictions, as a means to recover from potentially inadmissible thread interleavings. The local event executes a local action.

We define the semantics of these events in the following. For rules where a transaction t is defined in the prestate, its corresponding local configuration is denoted (c_t, σ_t, L_t). We utilize helper function $\mathsf{serpref}\colon\ \mathcal{L} \to \{\mathsf{true}, \mathsf{false}\}$ to (conservatively) determine whether a given shared log is the prefix of some serializable execution log. Finally, we define helper function $\mathsf{ref}\colon\ \mathcal{T} \to \mathcal{S}$ that — given a transaction t — retrieves the system state immediately preceding t's start.

The events are defined as follows:

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

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

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

$$\mathsf{corr}\ t \quad \frac{t \in T, \mathsf{ref}\ t \rightsquigarrow (T, \mu', \sigma, L)}{(T, \mu, \sigma, L) \to (T, \mu[t \mapsto \mu'(t)], \sigma, L)}$$

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

Note that the corr rule only applies changes to the local configuration corresponding to the given transaction t. 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. In the local rule, $\mathbb{C}$ represents the transition relation for local operations.

3.3 Formal Guarantees

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

Proof Sketch. The cmt *event acts as a gatekeeper, demanding that the log prefix 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.

Note that 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 $[t \mapsto (\sigma'_t, c'_t, L'_t)] \in \mu'$.

Example 1 (Self corrective synchronization). 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*. Post 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). *Assume that (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 (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* 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. Further assume that throughout any run of the system, only boundedly many transactions are created (via a* bgn *event), and these transactions do not have infinite executions. Then termination is guaranteed.*

Proof Sketch (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. In this Section, we define the thread-local concrete semantics of this language instantiating $\mathbb{C}$ of the local rule introduced in Section 3.2. Following the 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.

4.1 Language

We focus our formalization on the following language fragment:

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

This fragment captures a representative set of operations of the Java 7 `java.util.concurrent.ConcurrentMap` class.[7] 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 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 supports a standard `assert(b)` statement that let the execution go through iff the given Boolean condition holds. In particular, the language supports checking if a variable is `null`, and if the map contains a key. This statement is necessary to support conditional and loop statements.

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

4.2 Concrete Domain and Semantics

First of all, we instantiate the state of a transaction t introduced in 3.1 to the language of Section 4.1.

Let Var and Ref be the set of variables and references, respectively. Keys and values are identified by concrete references, and we assume `null` is in Ref. We define by $\mathsf{Env} : \mathsf{Var} \to \mathsf{Ref}$ the environments relating local variables to references. A map is then represented as a function $\mathsf{Map} : \mathsf{Ref} \to \mathsf{Ref}$, 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, $\langle \wp(\Sigma), \subseteq, \cup \rangle$.

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

Fig. 3: Concrete semantics

Figure 3 defines the concrete semantics. For the most part, the concrete semantics formalizes the API specification of the corresponding Java method. Note that `assert` is defined only on the states that satisfy the given Boolean conditions. In this way, the concrete semantics filters out only the states that might execute a branch of an `if` or `while` statement.

4.3 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 $\mathsf{isSummary} : \mathsf{HeapNode} \to \{\mathsf{true}, \mathsf{false}\}$ is provided; $\mathsf{isSummary}(n)$ returns true if n might represent many concrete nodes (that is, it is a summary node). We define by $\mathsf{Env} : \mathsf{Var} \to \wp(\mathsf{HeapNode})$ 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 $\mathsf{Map} : \mathsf{HeapNode} \to \wp(\mathsf{HeapNode})$, 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. Consider the motivating example in Figure 1. Abstract state $([\texttt{name} \mapsto \{n_1\}], [n_1 \mapsto \{\texttt{null}\}])$ represents that the key `name` is not in the map, while $([\texttt{name} \mapsto \{n_1\}], [n_1 \mapsto \{n_2\}])$ represents that it is in the map, and it is related to some value n_2. Finally, $([\texttt{name} \mapsto \{n_1\}], [n_1 \mapsto \{\texttt{null}, n_2\}])$ represents that `name` (i) might not be in the map, *or* (ii) is in the map related to value n_2.

4.4 Concretization function.

We define the concretization function $\gamma_\Sigma : \Sigma \to \wp(\Sigma)$ 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, $\gamma_{\mathsf{Ref}} : \mathsf{HeapNode} \to \wp(\mathsf{Ref})$. We assume that this concretization function concretizes `null` into itself ($\gamma_{\mathsf{Ref}}(\texttt{null}) = \{\texttt{null}\}$), and that it is coherent w.r.t. the information provided by $\mathsf{isSummary}$ ($\neg\mathsf{isSummary}(n) \Leftrightarrow |\gamma_{\mathsf{Ref}}(n)| = 1$).

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 pointwisely the concretization of environments and maps. This is formalized as follows:

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

Lemma 2 (Soundness of the domain). *The abstract domain is a sound approximation of the concrete domain, that is, they form a Galois connection [2]. 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 [3]).*

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

4.5 Abstract Semantics

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

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

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

$$\mathbb{S}[\![\texttt{v = m.putIfAbsent(k, v)}, (e, m)]\!] = (\pi_1(\mathbb{S}[\![\texttt{v = m.get(k)}, (e, m)]\!]), m') :$$

$$m' = \begin{cases} m[n \mapsto e(\texttt{v})] & \text{if } e(\texttt{k}) = \{\texttt{n}\} \wedge m(n) = \{\texttt{null}\} \\ m[n \mapsto m(n) \cup e(\texttt{v}) : n \in e(\texttt{k})] & \text{if } \texttt{null} \in m(n) \wedge |m(n)| > 1 \\ m & \text{otherwise} \end{cases} \quad (\texttt{pIA})$$

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

$$\mathbb{S}[\![\texttt{v = null}, (e, m)]\!] = (e[v \mapsto \{\texttt{null}\}], m) \quad (\texttt{nlas})$$

$$\mathbb{S}[\![\texttt{assert(x == null)}, (e, m)]\!] = \begin{cases} (e[\texttt{x} \mapsto \{\texttt{null}\}], m) & \text{if } \texttt{null} \in e(\texttt{x}) \\ \bot & \text{otherwise} \end{cases} \quad (\texttt{null})$$

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

$$\mathbb{S}[\![\texttt{assert(m.containsK(k))}, (e, m)]\!] = \begin{cases} \bot & \text{if } \forall n \in e(\texttt{k}) : m(n) = \{\texttt{null}\} \\ (e, m[n \mapsto m(n) \setminus \{\texttt{null}\}]) & \text{if } e(\texttt{k}) = \{n\} \wedge \neg\mathsf{isSummary}(n) \wedge \\ & \qquad m(n) \neq \{\texttt{null}\} \\ (e, m) & \text{otherwise} \end{cases} \quad (\texttt{cntK})$$

$$\mathbb{S}[\![\texttt{assert(!m.containsK(k))}, (e, m)]\!] = \begin{cases} \bot & \text{if } \forall n \in e(\texttt{k}) : \texttt{null} \notin m(n) \\ (e, m[n \mapsto \{\texttt{null}\}]) & \text{if } e(\texttt{k}) = \{n\} \wedge \neg\mathsf{isSummary}(n) \wedge \texttt{null} \in m(n) \\ (e, m) & \text{otherwise} \end{cases} \quad (\texttt{!cntK})$$

Fig. 4: Formal definition of the abstract semantics

Figure 4 formalizes the abstract semantics of statements and Boolean conditions, that, given an abstract state (as defined in Section 4.3) and a statement or Boolean condition of the language introduced in Section 4.1, 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 [2], this operational abstract semantics is the basis for computing a fixpoint over a CFG representing loops and conditional statements. 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 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. (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 {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. (pIA) updates the map like (put) but only if the updated key node might have been absent, that is, when $\texttt{null} \in m(n)$. (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, (nlas) relates the given variable to the singleton {null}. 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 {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.

Lemma 3 (Soundness of the semantics). *The abstract semantics is a sound approximation of the concrete semantics. Formally,* $\forall \texttt{st}, (e, m) \in \Sigma : \gamma_\Sigma(\mathbb{S}[\![\texttt{st}, (e, m)]\!]) \supseteq \mathbb{C}[\![\texttt{st}, \gamma_\Sigma(e, m)]\!]$*, where* $\mathbb{C}$ *represents the pointwise application of the concrete semantics introduced in Section 4.2 to a set of concrete states.*

Proof Sketch (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() in Figure 1, 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 (get).

Then the semantics of the Boolean condition of the if statements at line 3 applies rule (null) (that does not modify the abstract state) since conv is null, and we assumed create is true. Lines 4 and 5 applies rules (new) and (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 (!null) to σ (that is, $\bot$) 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 Transaction System Semantics

In this Section, we apply the abstract semantics $\mathbb{S}$ defined in Section 4.5 to infer corrective targets.

In our approach, we support a restricted transactional model. In particular, we assume that there are n transactions that start the execution together, each transaction commit only once, and all the transactions commit together at the end of the execution. Thanks to these assumptions, we can define a system that perform a *global* corrective synchronization at the end of the execution.

5.1 Serialized CFG

We apply the abstract semantics defined in Section 4.5 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 some 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 transactions.

Let $\{c^1, ..., c^k\}$ be the code of k different transactions. For each transaction type i, we create three static transaction identifiers t_1^i, t_2^i, and t_n^i. t_1^i and t_2^i represent precisely two concrete instances of c^i, while t_n^i 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_1^i, (ii) then t_n^i inside a non-deterministic loop (to overapproximate many instances of c^i)), and (iii) finally t_2^i. 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 $\mathcal{T}^\# = \{t_j^i : i \in [1..k], j \in \{1, 2, n\}\}$. Then our semantics on a serialized CFG returns a function in $\Phi : \mathcal{T}^\# \to \Sigma$.

Running example. Starting from the code in Figure 1, we build up a serialized CFG where all the transactions share the same key `name` (and in this way we target a scenario where conflicts might arise), and have `create` sets to `true` for the sake of presentation. This serialized CFG is made by the sequence of transactions $t_1; t_n^\star; t_2$ where $t_n^\star$ represents that the code of t_n is inside a loop.

Suppose now to analyze this serialized CFG starting from the abstract state $([\texttt{name} \mapsto \{n_1\}], [n_1 \mapsto \{\texttt{null}\}])$. The abstract semantics defined in Section 4.5 computes the following abstract poststate: $([\texttt{name} \mapsto \{n_1\}, \texttt{conv}_1 \mapsto \{n_1^1\}, \texttt{conv}_n \mapsto \{n_1^1\}, \texttt{conv}_2 \mapsto \{n_1^1\}], [n_1 \mapsto \{n_1^1\}])$ (where n_b^a represents the a-th node instantiated by transaction $\texttt{t}_\texttt{b}$, and $\texttt{conv}_c$ represents the local variable `conv` of transaction $\texttt{t}_\texttt{c}$). 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.

5.2 Extracting Possible Corrective Targets

First of all, notice that, given a transaction t, the corr rule of the transition system introduced in Section 3.2 requires that the state the system corrects to is reachable starting from the state at the beginning of the execution of t (retrieved by ref t) producing the same shared log (formally, ref $t \rightsquigarrow (T, \mu', \sigma, L)$). 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, $\forall t \in T : \text{ref } t = \sigma_0$) . 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 ref $t \rightsquigarrow (T, \mu', \sigma_0, \epsilon)$.

We compute possible corrective targets on the serialized CFG tc (as defined in 5.1) 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 : $\Sigma \rightarrow \{\text{true}, \text{false}\}$ that, given an abstract state, holds iff it represents and exact concrete state. Formally,

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

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

Proof Sketch (Proof Sketch). By definition of single, $\neg$isSummary(n) *for all the nodes* n *in* e *or* n*. By definition of* isSummary *we have that* $|\gamma_{\text{Ref}}(n)| = 1$*. 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 $\forall \phi \in S : \text{single}(\phi)$, 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: $\text{corrTarg}(\mathsf{T}, \mathsf{S}) = [\phi \mapsto \{\phi' : \phi' \in \mathbb{S}[\![tc, \phi]\!] \wedge \text{single}(\phi')\} : \phi \in S]$.

Running example. Starting from the entry state $([\texttt{name} \mapsto \{n_1\}], [n_1 \mapsto \{\texttt{null}\}])$, the exit state computed by the abstract semantics (as described in Section 5.1) is $([\texttt{name} \mapsto \{n_1\}, \texttt{conv}_1 \mapsto \{n_1^1\}, \texttt{conv}_n \mapsto \{n_1^1\}, \texttt{conv}_2 \mapsto \{n_1^1\}], [n_1 \mapsto \{n_1^1\}])$. 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.

5.3 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 can now prove that the corrective targets computed by corrTarg satisfies the ref conditions as imposed by corr rule.

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 (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 $\gamma_{\Sigma}(\phi_0) = \{\sigma_n\}$. 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 poststate, 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 poststate, 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 poststate 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 Implementation and Evaluation

In this section, we first describe our prototype implementation of the overall system (both the static analysis and the runtime synchronization algorithm), and then its experimental evaluation over a set of four real-world subjects.

6.1 Prototype System

We have created a Java implementation of our static analysis for composed Map operations, as described in Section 4. Given n types of transactions, our implementation builds a serialized CFG as described in Section 5.1, and then it computes a fixpoint over it relying on the abstract semantics formalized in Section 4.5. We support all the operations listed in Section 4.1. Running times are negligible compared to the rest 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 prestates to sets of poststates that are obtainable via serializable execution of the transactions from the prestate. 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 prestate where k was not in the map and the value passed to the function was v, then in the poststate the key k is made to point to the value v pointed-to by the second argument v in the prestate. The runtime system S is parameterized by the specification, which it loads at the beginning of the concurrent run.

As discussed in Section 5.3, in our current prototype all transactions are assumed to start simultaneously, which is useful for the common case of loop parallelization. Each concrete transaction is mapped to its abstract counterpart, as explained in Section 5.1. 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 current prototype, we limit transactions to a single commit point before completion. Corrective synchronization occurs only on failed commits (as discussed in Section 3.3), in which case the transaction's shared log, local state and return value (if it exists) are all (potentially) modified according to the corrective specification.

6.2 Methodology and Experimental Setup

We conducted our experiments on server with two Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz (16 cores) CPUs with 132GB of RAM. The server runs version 6.6 of the CentOS Linux distribution, and has the Sun 64-Bit 1.8 Java virtual machine (JVM) installed.

In our experiments, we varied two aspects of parallel execution: *workload size*, measured as the number of transactions executed by a given thread for a fixed number n of threads, and *concurrency level*, setting the number of threads for a fixed workload size. For the former, we fixed $n = 8$ and varied the number of transactions, where each transaction executes a single client method. For the latter, we experimented with 2 to 32

```
Object internalDeserialize(PofReader in) {
  Class t = in.getPofContext().getClass(in.getUserTypeId());
  ObjectFormat f = formats.get(t);
  if (f == null) { f = new ObjectFormat(t); formats.put(t, f); }
  return resolve(f.deserialize(in)); }

void internalSerialize(PofWriter out, Object o) {
  Object v = replace(o); Class t = v.getClass();
  ObjectFormat f = formats.get(t);
  if (f == null) { format = new ObjectFormat(t); formats.put(t, f); }
  format.serialize(out, v); }
```

Fig. 5: Concurrent methods from the Gridkit benchmark

threads with workload size fixed at 10. For statistical soundness, we report the average performance results across 10 independent repetitions of each experiment.

We compare our technique against two competing solutions: (i) a pessimistic concrete-level variant of STM, as available via version 1.3 of the Deuce STM (the latest version)[8], and (ii) a lock-based synchronization algorithm boosted with Map semantics [8], such that the locks are of the same grain as their corresponding abstract locks in boosted STM.

6.3 Subjects

Our benchmark suite consists of four subjects, all of which are taken from popular open-source code bases and have been used in past studies on concurrency [21,22]. Apache Tomcat is web-app container in wide deployment. Within it, we consider ApplicationContext, which is a concrete representation of a web application's execution environment. dyuproject is a framework for development of Java web applications. We experiment with StandardConvertorCache, which handles conversion of objects to/from JSON format. Flexive is a next-generation content repository for development of web apps, in particular in an enterprise setting. We have included FxValueRendererFactory from it, which is responsible to render transparently certain objects in a language. Finally, Gridkit is a library containing utilities related to in-memory data grids. In it, ReflectionPofSerializer is a concurrent data structure that supports generic POF serialization via reflection.

We have extracted two concurrent methods out of each benchmark. These were selected based on past studies that report on concurrency bugs in, and interference between, the selected methods [21]. As an illustration, in Figure 5 we present the method pair from Gridkit. In both of these methods, the intended atomic "put-if-absent" behavior (available as the putIfAbsent API) is misimplemented due to the separation between the get and put calls. (We omit other method pairs for lack of space.)

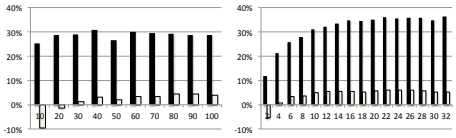

Fig. 6: Apache Tomcat ApplicationContext (l: workload; r: conc)

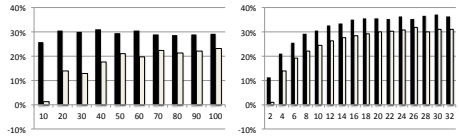

Fig. 7: dyuproject StandardConvertorCache (l: workload; r: conc)

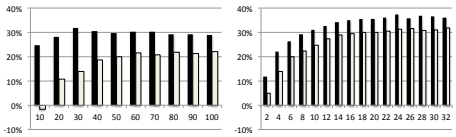

Fig. 8: Flexive FxValueRendererFactory (l: workload; r: conc)

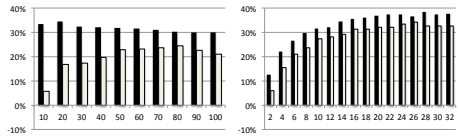

Fig. 9: Gridkit ReflectionPofSerializer (l: workload; r: conc)

6.4 Performance Results

We depict the performance results in Figures 6-9, where for each benchmark we plot the results when increasing the size of the workload while fixing the number of threads (on the left), and when varying the number of threads (on the right). Each plot presents a two column diagram, corresponding to STM and corrective synchronization (left/black and right/gray, respectively), which indicates the relative gain thanks to these approaches in comparison with lock-based synchronization. Indeed, the lock-based solution is overly conservative, yielding roughly a linear trend line of decrease in performance as either workload size or the number of threads increases.

The aggregate performance results, averaging across all workloads and concurrency levels, are as follows:

	STM		corrective synchronization	
	workload	conc.	workload	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]. As the numbers indicate, corrective synchronization achieves better performance than STM with relative improvement on locks that is more than x2 better than STM.

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

6.5 Discussion

An interesting observation w.r.t. the raw performance results is that while corrective synchronization provides stable and consistent performance improvement across all benchmarks, STM seems to do much worse on Tomcat compared to all other benchmarks. We have analyzed this gap. The reason for this discrepancy,is that in both dyuproject and Flexive and Gridkit, once a first transaction manages to update the value corresponding to the input key, all other (nonoverlapping) transactions effectively become read-only transactions, and so conflict free. Hence STM achieves significant improvement compared to locks. In the case of Tomcat, however, this pattern does not hold. Transactions throughout the entire run perform write operations, leading to conflicts that degrade the performance of STM.

Interestingly, even on the benchmarks other than Tomcat STM is not able to compete with the low overhead, and thus better performance, of corrective synchronization. We suspect that the reason is that STM has to instrument memory accesses, whereas corrective synchronization applies state corrections directly.

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 based on static program analysis.

Rollback optimizations. 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 have been proposed along each of these directions.

Conflict detection is primarily based on violations of disjoint parallelism. That is, if two transactions perform simultaneous read/write or write/write accesses to the same memory location, then they are deemed conflicting. An effective approach to mitigate false conflicts is to leverage the built-in guarantees of concurrent data structures [8,15,25]. As an example, two ConcurrentHashMap instances that perform put operations involving different keys may still appear to conflict as they both access the size field.

Transactional boosting [8] is a systematic methodology to specify the behavior of linearizable data-structure operations in terms of their semantic footprint. The Galois system [15], follows a similar approach with emphasis on the Graph ADT. The Hawkeye tool [25] detects impediments to parallelization accurately by mapping between the concrete and abstract representations of the data structure (albeit not its operations).

In another study, Tripp et al. [24] mine commutative behaviors involving *multiple* operations out of execution traces of the program that appear to conflict when viewed at the granularity of individual operations. These are then translated into logical conditions used to avoid spurious conflicts in lazy STM.

Another approach to reduce conflict is to leverage available nondeterminism [23]. If a transaction can choose between different nondeterministic behaviors (e.g., selecting

among different paths between two graph nodes), then conflict is potentially reduced by directing the transaction toward a path where interference with other threads is less likely.

A well-known solution to restrict the extent to which a transaction rolls back is checkpointing [12,4]. Checkpointing introduces the notion of a partial abort, where the transaction resumes from some intermediate point rather than fully resetting its execution and effects. Nested transactions [17,1] achieve a similar effect in that only the nested transaction, but not its parent transaction, aborts and restarts. Yet another mechanism to reduce rollback overhead is elastic transactions [5], which avoid wasted work by splitting into multiple pieces.

Finally, we note the Push/Pull model [13], which also makes use of local/shared logs, and is flexible enough to express rollback-based transactions. However, corrective synchronization is not expressible in Push/Pull. We introduce a new rule, corr, and treat cmt differently with the serpref function.

Use of static analysis. 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. We survey some of these works in the following.

Golan-Gueta et al. [6] 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 enables more granular synchronization compared to the worst-case assumption that the transaction may perform any action in its future.

Autolocker [16] applies static analysis to determine a correct locking policy, free of deadlocks and race conditions, given (i) a specification of pessimistic atomic sections and (ii) annotations that connect locks to shared sections. The analysis, encoded as a type system, guarantees deadlock and race freedom.

Hawkins et al. [7] propose a system that transforms a program consisting, in part, of concurrent relations into a program where the relations are represented as concrete concurrent data structures, as well as locks, that together ensure correct synchronization by construction. They guarantee serializability and deadlock freedom.

Prountzos et al. [19] optimize the Galois system [15] via static shape analysis [20]. They apply the TVLA system to identify fail-safe points during the transaction's execution, beyond which neither backup nor conflict detection are necessary.

8 Conclusion and Future Work

We have presented an alternative to the lock- and retry-based synchronization methods, which 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, and experimented with, an instantiation of corrective synchronization for composed operations over ConcurrentMaps, where correct states are computed via offline static analysis. Our initial results show promise.

In the future, we intend to define, and experiment with, more variants of corrective synchronization. As part of that, we plan to 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. 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.
3. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. Maurice Herlihy and Eric 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'08)*, 2008.
10. 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.
11. Andi Kleen. Scaling existing lock-based applications with lock elision. *Commun. ACM*, 57(3):52–56, 2014.
12. 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.
13. 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.
14. 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.
15. 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.

16. 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.
17. 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.
18. Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979.
19. 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.
20. Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24:217–298, 2002.
21. 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.
22. 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.
23. 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.
24. Omer Tripp, Roman Manevich, John Field, and Mooly Sagiv. JANUS: exploiting parallelism via hindsight. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 145–156, 2012.
25. 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.