Static Mining of Common Concurrency Patterns

Evgeny Abramovich
Static Mining of Common Concurrency Patterns

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Evgeny Abramovich

Submitted to the Senate of the Technion — Israel Institute of Technology
Nisan 5775 Haifa March 2015
This thesis was completed under the supervision of Prof. Eran Yahav (Department of Computer Science, Technion)

I would like to thank my supervisor, Prof. Eran Yahav, for his support, his will to provide the best conditions possible and the endless positive attitude throughout the way.

Thanks also to Prof. Danny Raz, who by fulfilling my unusual requests made it possible for me to start my research, to Alon Mishne for the help at the beginning of my work and finally to little Talya, for being a calm gorgeous baby and letting me finish this work.

The generous financial support of the Technion is gratefully acknowledged. This work was partially supported by the Israeli Science Foundation (ISF), grant number 965/10.
# Contents

Abstract ................................................................. 1

1 Introduction ......................................................... 2

2 Overview .............................................................. 8
   2.1 Concurrent Data Structure Real World Example .......... 8
   2.2 Concurrency Pattern Types .................................. 8
      2.2.1 Locking patterns ......................................... 8
      2.2.2 Lock-Free patterns ....................................... 12
   2.3 Our Approach .................................................. 13

3 Background .......................................................... 18

4 Static Mining of Concurrency Patterns ...................... 20
   4.1 Instrumented Semantics Tracking Partial Specifications . . 20
   4.2 Abstractions for Partial Programs ......................... 23

5 Experimental Evaluation ......................................... 27
   5.1 Prototype Implementation ................................... 27
   5.2 Results ....................................................... 28
      5.2.1 Locking Patterns Study ................................ 28
      5.2.2 Concurrency Patterns (including locks) ............... 28
      5.2.3 Discussion ................................................ 36

6 Related Work ....................................................... 40

7 Conclusion .......................................................... 44
## List of Figures

1.1 Different candidate implementation for a put-if-absent operation on a map .......................... 3  
2.1 Example - interaction between 2 data structures with locks  . 9  
2.2 Example - compute if absent ................................. 10  
2.3 Example - compute until success .............................. 11  
2.4 Atomicizable code ............................................ 13  
2.5 Atomic usage example of ConcurrentHashMap ............... 14  
2.6 Non-atomic example usage of ConcurrentHashMap ............ 15  
2.7 Analysis Flow .................................................. 15  
4.1 Example - states throughout a method analysis ............... 24  
4.2 Example - abstract history creation .......................... 26  
5.1 Two Method Sequence .......................................... 28  
5.2 Two Method Sequence Modified ................................ 28  
5.3 Global Lock Example .......................................... 29  
5.4 Single Method Lock Example .................................. 29  
5.5 Pattern 1 - Atomic write after read .......................... 30  
5.6 Pattern 2 - Non-atomic write after read - no null check on putIfAbsent ........................................ 30  
5.7 Pattern 3 - Non-atomic "put if absent" - usually using Map interface ................................................. 31  
5.8 Pattern 4 - Locked write after read (or multiple writes) .... 32  
5.9 Pattern 5 - Write after read without lock - except Pattern 3 .... 33  
5.10 Pattern 6 - A single concurrent write ........................ 33  
5.11 Pattern 7 - Atomic write after read (a variation) ............ 36
Abstract

We present a framework for static mining of synchronization patterns from large scale software repositories. The main idea is to use static analysis for extracting synchronization patterns from a large number of client programs using a concurrent component, and aggregating the extracted patterns to obtain the common synchronization patterns in the code base. These patterns can reveal both the correct usage and examples of incorrect usage to programmers using the concurrent component. Our analysis uses automata-based abstraction to capture sequences of API operations and synchronization operations. By combining information about the sequence of API calls with synchronization operations, we uncover common synchronization patterns used for the underlying components. We address both lock-based synchronization and lock-free patterns. We have implemented our approach in a tool called SynchMiner based on the WALA static analysis framework, and applied it to identify 6 major patterns by analyzing 23 complete real-world concurrent libraries and also different 10000 class files with concurrency.
Chapter 1

Introduction

A large portion of modern software is run concurrently by multiple threads. To enable such concurrent use, synchronization constructs and concurrent data structures were introduced [21, 29].

The usage of such constructs hides the low level implementation and provides powerful capabilities to the programmer. However, both: (i) applying of such constructs correctly throughout the code in every usage, and (ii) composing the constructs [27, 28] are exposed to multiple types of concurrency-related bugs.

Fixing these bugs is notoriously hard. Testing is difficult since the incorrect behavior can be hard reproduce in the testing environment. Traversing all the possible interleavings leads to a computational explosion and is infeasible for real world test cases. Furthermore, even if testing would be feasible, it is hard to define the exact desired specification.

Learning from experience of others. A way to avoid these bugs is to rely on code examples of other programs using the synchronization constructs. Widely used open source libraries are available and contain multiple synchronization solutions. The vast amount of code available on the web is increasing on a daily basis. Open-source hosting sites such as GitHub contain billions of lines of code. The availability of code search services (e.g., Krugle Open Search [2], Black Duck Open Hub Code Search [14], GitHub Search [12]) also expose the programmer to a vast number of API usage examples in concurrent environment. The examples are typically available in the form of complete files from open-source projects.
Figure 1.1: Different candidate implementation for a put-if-absent operation on a map

In addition to the vast amounts of available source code, there is also a vast number of usage examples only available in binary form (in our setting, this means Java bytecode). Mining representative patterns requires analyzing both sources and binaries, statically extracting patterns, and aggregating them into a form that could be presented to the programmer.

**Data structures in concurrent environment - basic usage example.**

Fig. 1.1 shows three implementations of the same method. The method’s semantics is “put if absent”, which means: given a key and a value, associate the value with the key if and only if no value is associated with the key, and return the value associated with the key before the operation (which may be "none"). This method may be used in a sequential non-concurrent execution, but mostly used in shared map implementations. In concurrent use:

- The first method may not follow the intended method’s semantics, two
concurrent threads may both successfully add value to the map, and the second put won’t comply to the “put if absent” specification.

- The second method does follow the semantics, the value will be added to the structure only if the key doesn’t have a value associated to it. The main problem with this approach is that it locks the entire data structure (In this example, by a caller designated method lock) on each call which has significant negative effect on performance.

- The third method uses the `putIfAbsent` method, which is a member of the `java.util.concurrent.ConcurrentMap` interface. Its specification (from its Javadoc [1]): if the specified key is not already associated with a value, associate it with the given value; returns the previous value associated with the specified key, or null if there was no mapping for the key. This simple method uses `putIfAbsent` correctly, and follows the intended semantics. However, `putIfAbsent` can be and is widely misused as shown further in the thesis.

**Code Patterns.** From the usage examples, code patterns may be extracted. A pattern corresponds to an abstract history of method invocations on an abstract object.

- An abstract history is an history that possibly describes more than a single trace of events (method invocations / locks / return value checks as described in Section 4).

- An abstract object is an object that possibly represents more than one concrete object in the program or no real concrete object.

A possible use of this pattern detection can be labelling the originated method or code segment as “non-atomic”. Using the label, the programmer can check herself that the labelled code is not executed concurrently, or dynamic analysis can check if it runs concurrently and issue an alert.

An example of such pattern in the second example is a simple sequence get → put with a “No-lock” indication on an object referenced by `map`. This pattern is violating the atomicity of the composed operation “put if absent”. Thus, it cannot be used in concurrent executions, where atomicity is required.
The patterns may be richer - for example, consider the method return values and the parameters. Judicious middleground needs to be found between non-useful detail-lacking pattern types and too detail-rich patterns which cannot be summarized.

**Goal.** The goal of our work is to mine usage patterns of concurrent objects from client programs using the concurrent objects (in most cases, our concurrent objects are the standard Java implementations of concurrent data structures). Mining such patterns requires handling the following challenges:

- **Static analysis:** because the code examples that we obtain are often partial (e.g., a single file from a project), we cannot run the examples, and often cannot even directly compile them. This motivates the use of static analysis for extracting the specification from each individual example.

- **Track object method invocation histories of a single object:** In our static analysis, we have to track how the concurrent object(s) are being used. As we will see later, we do that by introducing an instrumented semantics (Section 4.1) that augments the program state by tracking the history of method invocations for any object of interest.

- **Handle unbounded number of objects and unbounded histories:** The static analysis has two sources of unboundedness. First, the number of dynamically heap-allocated objects does not have an a priori bound. This means that the analysis has to track a potentially unbounded number of objects. Second, due to loops in the program, the number of events in a history is also potentially unbounded.

- **Handle concurrency-related structures:** In addition to tracking the history of method invocations, we have to track the use of synchronization primitives. For example, we have to track synchronization operations due to monitor-enter and monitor-exit (the use of Java’s `synchronized`), as well as other synchronization operations.

**Our approach.** We use a technique for statically mining concurrency API specifications from multiple examples of API’s clients. It is based on the work by Mishne et al. [19]. This technique is enriched by concurrency-related constructs, such as monitor detection. For better precision of the
mining, we introduced an *interprocedural support* of the specification mining. We decided not to support recursion in our analysis.

These specifications are filtered and summarized in order to provide a wide pattern usage picture in the given examples.

In the long term, the analysis results usage possibilities are

- Finding subtle patterns through interprocedural analysis.
- Anomaly detection - when common and widespread patterns are identified, other patterns may indicate misuse.
- Troublesome pattern detection - detection of patterns corresponding to known misbehaviors.
- Categorizing patterns for detection of similar ones, proposing a best implementation given constraints.
- Classifying patterns into families, similar to Atomic Write after Read and Read After Write, investigated in [5]. The thesis describes mandatory usage of pattern families for correct synchronization.

**Related Work** Here, we only mention a few closely related works. Additional related work is reviewed in section 6.

Our work was inspired by the work of Schaham et al. [27]. They use a combination of static and dynamic analyses. We expected to detect the malfunctions described in this work by static analysis only, and indeed found patterns (Fig. 5.6, Fig. 5.10 and Fig. 5.9 in section 5) corresponding to linearizability violations.

Our work has common elements with the work of Mishne, Shoham and Yahav [19] of temporal specification mining from multiple examples. The main differences are: (i) the synchronization constructs in the specification - leading to tracking a larger set of related objects and to richer specification event set; and (ii) native support of interprocedural analysis.

**Main Contributions** The contributions of this thesis are:

- We present a novel framework for interprocedural specification mining of concurrency related specifications from API usage in Java libraries. The mined specifications are generalized typestate properties
either containing synchronization constructs or specific to common concurrently-used data structures.

- We implemented our approach in a tool called SynchMiner, and demonstrated its use by analyzing dozens of commonly used libraries and thousands of small partial code examples. We show that SynchMiner is able to mine interesting usage patterns.
Chapter 2

Overview

2.1 Concurrent Data Structure Real World Example

Fig. 2.1, Fig. 2.2 and Fig. 2.3 show examples of concurrent usage of data structures. Their usage should seem atomic in every possible interleaving.

Fig. 2.1 shows an example of an API usage with external locking, which is the use of Java monitors (via synchronized, etc.,) or other lock structures. In the example, two locks for data structures are being acquired for a consistent read of their contents.

Fig. 2.2 and Fig. 2.3 present usages of a dedicated concurrent map - a sequence of map operations. Both of these usages are free of external locking. Each of these figures contains the relevant specification automaton revealing the concurrency related details of the method.

2.2 Concurrency Pattern Types

2.2.1 Locking patterns

A locking pattern is an access to a data structure using a lock. Types of locks (in Java, thus available to us) are:

- synchronized methods
- synchronized blocks
synchronized(exceptionPages) {
  synchronized(statusPages) {
    ErrorPage results1[] = new ErrorPage[exceptionPages.size()];
    results1 = (ErrorPage[]) exceptionPages.values().toArray(results1);
    ErrorPage results2[] = new ErrorPage[statusPages.size()];
    results2 = (ErrorPage[]) statusPages.values().toArray(results2);
    ErrorPage results[] = new ErrorPage[results1.length + results2.length];
    for (int i = 0; i < results1.length; i++)
      results[i] = results1[i];
    for (int i = results1.length; i < results.length; i++)
      results[i] = results2[i - results1.length];
    return results;
  }
}

Figure 2.1: Example - interaction between 2 data structures with locks
public static AndroidTools forOsFamily(String osFamily) {
    AndroidTools instance = androidTools.get(osFamily);
    if (instance == null) {
        AndroidTools newInstance = null;
        if (osFamily.equals("windows")) {
            newInstance = new WindowsAndroidTools();
        } else if (osFamily.equals("unix")) {
            newInstance = new UnixAndroidTools();
        } else {
            throw new UnsupportedOperationException("Don’t know how to start android tools on "+osFamily);
        }
        instance = androidTools.putIfAbsent(osFamily, newInstance);
        if (instance == null) instance = newInstance;
    }
    return instance;
}

Figure 2.2: Example - compute if absent
```java
void inc(Class<? super T> key) {
  for (; ; ) {
    Integer i = get(key);
    if (i == null) {
      if (putIfAbsent(key, 1) == null) // LP if succeeds
        return;
    } else {
      if (replace(key, i, i + 1)) // LP if succeeds
        return;
    }
  }
}
```

Figure 2.3: Example - compute until success
• condition variables (wait(), notify())
• implementations of java.util.concurrent.locks.Lock interface.

Our focus in the locking pattern scope is on synchronized blocks and methods.

The locking patterns can be clustered to the following types:

• Global lock - a whole sequence of accesses to a data structure is made under a lock.

• Global lock on an "atomicizable" sequence. Fig. 2.4, assuming that the three blocks are the only accesses to sessions fields, it can be immediately switched to ConcurrentHashMap. The first two blocks have exact matches in ConcurrentHashMap. Block 3 doesn’t have an exact match, but the usage of two operations is wrong - only remove suffices - its return value should be used.

• Optimistic lock - a read without locks followed by a locked write. This probably indicates synchronization problems.

• No lock - a sequence without locks.

2.2.2 Lock-Free patterns

A concurrency pattern is a sequence of calls on a concurrent data structure (e.g. java.util.concurrent.ConcurrentHashMap), having a property, such as atomicity. The following are concurrency pattern examples from Shacham et al. [27]. After investigating multiple code examples, we found these were the most common lock-free patterns of ConcurrentHashMap.

These patterns represent an attempt of the "compute if absent" flow, which is: given a key, find the mapped value; if no value is mapped, compute a new value and map it to the key. An atomicity problem may occur when 2 threads do not find any value and both compute it at the same time. The correct atomic usages of "compute if absent" followed the pattern in Fig. 2.5, while bugs often fell into Fig. 2.6’s pattern.

• Fig. 2.5 - atomic pattern. A null check of the putIfAbsent’s return value is present here. All the threads attempting to map a value perform
```java
// block 1
synchronized (sessions) {
    for (Iterator<Map.Entry<String, Session>> iter = sessions.entrySet().iterator(); iter.hasNext();)
        Session s = iter.next().getValue();
    if (...) {
        ...
        iter.remove();
    }
}

// block 2
synchronized (sessions) {
    sessions.put(id, s);
}

// block 3
synchronized (sessions) {
    sessions.get(id).dispose();
    sessions.remove(id);
}
```

Figure 2.4: Atomicicizable code

a null check. The thread that succeeds to map the value returns the value it created, other threads return the value created by the thread that succeeded.

- Fig. 2.6 - non atomic pattern. No null check, and a thread may return an object that was not mapped to the key.

### 2.3 Our Approach

Fig. 2.7 shows an overview of our analysis.

SynchMiner analyzes compiled library binaries. They binaries can be existing compiled libraries used in the real world or "pseudo libraries" - multiple source files compiled together by a partial compiler into a Java library. These partial programs aim to represent all single usages of each accessible API method from the library.

**Obtaining partial API specs.** SynchMiner statically analyzes the binaries, and tracks the sequences of API method invocations observed in them for each API object, using automata. In the evaluation section, there are code examples with the resulting DFAs, e.g. Fig. 5.5. There are several points to observe here:
1   Namespace findOrCreate(Symbol name)
2       Namespace ns = namespaces.get(name);
3       if(ns != null)
4           return ns;
5       Namespace newns = new Namespace(name);
6       ns = namespaces.
7           putIfAbsent(name, newns);
8       return ns == null ? newns : ns;
9   }

Figure 2.5: Atomic usage example of ConcurrentHashMap
Class getKey(Annotation annotation) {
    if (annotation == null) {
        throw new NullPointerException (...);
    }
    Class clazz = lookup.get(annotation.getClass());
    if (clazz == null) {
        clazz = getClass(annotation);
        lookup.putIfAbsent(annotation.getClass(), clazz);
    }
    return clazz;
}

Figure 2.6: Non-atomic example usage of ConcurrentHashMap

Figure 2.7: Analysis Flow
Locks on a method. On each method, all the obtained and released locks (all monitor enters and exits) are indicated, as seen in the DFA in Fig. 5.3. The locks are acquired cross-history, meaning all relevant histories contain the lock indication, a lock event in the DFA.

Return values of API methods. A return value of an an interesting method is tracked for null comparisons. It can be seen in get method and in putIfAbsent method in Fig. 5.5. Every method returning an object reference can be tracked in such way. For noise reduction, we limit the tracked method set to these methods only.

Partial method sequences. In the absence of a clear entry point for an example, we treat each public library method as a possible entry point. Thus, the API methods tracked on each object usually do not reflect its creation. Similarly, a single example usually does not capture the full sequence of events in an object’s lifetime. That’s because the emphasis on concurrency-related patterns is on handling short sequences, when each sequence can run concurrently with any other sequence.

Single type. Some API usage scenarios span multiple objects of different types. We analyze a single type only.

Unbounded sequences and sets of allocated objects. The analysis has to address two sources of unboundedness: an unbounded length of API method invocation sequences (e.g., the for loop in Fig. 2.3) and unbounded number of allocated objects (Any public method may actually be run on an unbounded number of objects allocated elsewhere). To address the former, we introduce an abstraction representing sequences in a bounded way using DFAs, as described in Section 4. To address the latter, we use a heap abstraction based on access paths similar to the ones used in [11], combined with “may point to” analysis used by WALA.

Consolidation. To consolidate partial specifications, we use a few clustering techniques, specialized for partial histories.

Similarity Similar histories are merged. Histories are merged to the same cluster when they have the same set of method calls, the interesting parameters and the locks are isomorphic.

Method Semantics Similarity Given 2 methods which have similar known semantics, such as both are “read methods”, histories with the same method semantics are merged together.
Inclusion: All the histories within a cluster are merged, putting the pieces together where they fit best, like a puzzle, when the method similarity is as described above.
Chapter 3

Background

In this section, we provide basic background and definitions used throughout the thesis. We first define what we mean by the terms *API* and *client program*.

**Library API:** A library API is a collection of class names $T_1, \ldots, T_n$, where each class has an associated set of method signatures corresponding to methods provided by the class.

**Concurrent API** is an *API* of a collection class which intended to be accessed concurrently by many threads.

**Client:** We use the term *client program* of an API to refer to a program that uses a given API by allocating objects of API classes and invoking methods of the API classes. We refer to a method of the API as *opaque method* since the implementation details of such a method are unknown to the client. More importantly, we assume that such methods only affect the internal state of the API and do not change other components of the global state of the client program. In contrast, we refer to a method defined within the client program as *transparent method*.

**Partial Program** is a program using a single method as an entry point, when all the related objects are created using default constructors.

**Concrete Semantics.** We assume a standard imperative object-oriented language with native multithreadig support, and define a program state and evaluation of an expression in a program state in the standard manner. Restricting attention to reference types, the semantic domains are defined
in a standard way as follows:

\[
\begin{align*}
L^2 & \in 2^{\text{objects}^2} \\
v^2 & \in \text{Val} = \cup \{\text{null}\} \\
\rho^2 & \in \text{Env} = \text{VarId} \rightarrow \text{Val} \\
\pi^2 & \in \text{Heap} = \text{objects}^2 \times \text{FieldId} \rightarrow \text{Val} \\
\text{state}^3 & = (L^2, \rho^2, \pi^2) \in \text{States} = 2^{\text{objects}^2} \times \text{Env} \times \text{Heap}
\end{align*}
\]

where is an unbounded set of dynamically allocated objects, \( \text{VarId} \) is a set of local variable identifiers, \( \text{FieldId} \) is a set of field identifiers. To simplify notation, we omit typing from the definition. In practice, objects are typed and all components admit correct typing.

A program state keeps track of the set \( L^2 \) of allocated objects, an environment \( \rho^2 \) mapping local variables to values and a mapping \( \pi^2 \) from fields of allocated objects to values.

**Partial Programs.** Our approach requires handling of partial programs. To simplify presentation, we assume that the code of each method we inspect is known in full, which means that all modifications of local state are known. An inspected method may invoke unknown methods, and refer to unknown types and fields.

We assume a standard semantics of partial programs updating a program state \( (L^2, \rho^2, \pi^2) \), where an invocation of an unknown transparent method can allocate any number of fresh objects and can modify its reachable objects arbitrarily (e.g., [8]).
Chapter 4

Static Mining of Concurrency Patterns

The first step of our approach is analyzing each code example individually to extract a temporal specification describing how the use of concurrency in API is entangled in the example.

In this section, we first define an instrumented concrete semantics for partial programs that tracks "histories" for each tracked object, representing the way the API has been used. We show how partial specifications, in the form of histories, express partial information extracted from a partial program. Then, we describe an analysis in terms of an abstraction of the instrumented concrete semantics.

4.1 Instrumented Semantics Tracking Partial Specifications

We define an instrumented semantics that tracks the way a program uses an API by recording a history of relevant method invocations and other operations described later.

Method Call Events. We refer to the invocation of an opaque method as an event. An event has a receiver object and a method signature. The receiver is the concrete object whose method is invoked. We may differentiate the events by API function arguments, resulting in a more precise analysis or treat each invocation in the same way.
**Lock Events.** We refer to entering a monitor (e.g., synchronized block) as an event. The event has an object only - the one being locked. All the objects in objects are the event receivers, because the locked object can be different than the receiver of the API calls, and still be used to protect an API call sequence.

**Unlock Events.** Similarly we refer to exiting a monitor as an event on all objects in objects.

**Not Null Events.** On known concurrent methods of interest, return value is used in the flow. These methods’ semantice are usually READ such as \( x=m.\text{get}() \) or UPDATE, such as \( \text{replace}, x=m.\text{putIfAbsent}() \). We track the return value \( x \) for further null comparison. An example on such event used in our analysis is "the return value of a method \( M \) is not null".

**Histories.** In our instrumented semantics, each concrete object is mapped to a "concrete history" that records the sequence of events that has occurred for that object. Technically, we define the notion of a history as capturing a regular language of event sequences.

**Definition 4.1.1** Given a set of events \( \Sigma \), a history \( h \) is a finite automaton \( (\Sigma, Q, \text{init}, \delta, F) \), where \( Q \) is a set of states, init is the initial state, \( \delta : Q \times \Sigma \rightarrow 2^Q \) is the transition relation, and \( F \neq \emptyset \) is a set of final states.

We define the traces represented by \( h \), \( Tr(h) \), to be the language \( \mathcal{L}(h) \).

A concrete history \( h^c \) is a special case of a history that encodes a single finite trace of events, that is, where \( Tr(h^c) \) consists of a single finite trace of events. In Section 4.2 we use the general notion of a history to describe a regular language of event sequences. We refer to a history that possibly describes more than a single trace of events as an abstract history.

In principle, a history may be associated with different levels of a state, such as: (i) a single history may be associated with a global state and track the global sequence of events over all API objects. Because a global history maintains the order between all events, it may create artificial ordering between events of independent objects. (ii) a history may be associated with a single object in the state. Such per-object history does not capture any ordering between events on different objects, even when such relationship may be important (cf. Fig. 2.1 in Section 2).

In our work, each history is associated with a single object in the state, and the locking context is associated with the global state. In that way, the
The locking state of the global history is cloned and injected into each one of the single object histories. We refer to such histories as **per-object histories with locking context**.

**State with Locks.** Each state in \( \mathcal{Q} \) has a **lock stack** of locked objects. The stack corresponds to the locking order of all currently locked objects. Maintaining the locking order may assist in deadlock detection (using our SYNCHMINER tool, deadlocks were detected on small artificial examples only) and in assuring that the release order is the same as the acquisition order.

**Instrumented Semantic: State.** We denote the set of all concrete histories by \( \mathcal{H} \). We augment every concrete state \( h \in L \times \rho \times \pi \times \mathcal{H} \) with an additional mapping \( \text{his}^3 : L \times \rho \times \pi \times \mathcal{H} \) that maps an allocated object of a tracked type to its concrete history. A state of the instrumented concrete semantics is a tuple \( (L, \rho, \pi, \text{his}^3) \).

**Instrumented Semantics: Transformers.** The \( \text{his}^3 \) component of the concrete state tracks the histories for all tracked objects. It is updated as a result of allocation and calls to methods of a tracked type in a natural way. Here, we only show the treatment of special concurrency events and locking context of a history start:

- **Locks / Unlocks in code:** In case of a lock, such as monitor enter when entering synchronized block, or a respective unlock, all live histories are extended with a respective lock / unlock event and for each one of them, the current state’s lock stack is updated.

- **First Object Event on Locking Context:** The concrete history of an object always initialized to the acquiring of all locks in the "global locking stack" - the current locking stack of all live histories. This history is extended by a number of lock events as the number of objects in the stack, acquired in the order in stack, before other events.

  Since the semantics considers each public method as a potential entry point, we assume implicitly unknown prefixes/suffixes for the generated histories, unless an init event is observed. init “grounds” a prefix because we know that no event can precede it. Histories that start with any event other than init are considered **non-initialized**.

**Example 4.1.2** For the method depicted in Fig. 2.1 (non-initialized, like other major examples in the thesis), Fig. 4.1 shows a part of his^3 mappings.
during the analysis run. The histories are shown for the interesting objects only, instances of `java.util.HashMap`.

### 4.2 Abstractions for Partial Programs

The instrumented concrete semantics uses an unbounded description of the program state, resulting from a potentially unbounded number of objects and potentially unbounded histories. We now describe an abstract semantics that conservatively represents the instrumented semantics using a bounded program state, and provides a basis for our analysis. Specifically, the analysis propagates a sound approximation of program state which is based on a heap abstraction and a history abstraction. The heap abstraction is in fact quite subtle, but applies standard concepts from modular analyses (e.g., [8]).

**History Abstractions.** A concrete history simply encodes a sequence of events. A conservative abstraction for it can be defined in many ways, depending on the information we wish to preserve.

In practice, automata that characterize API specifications are often simple, and further admit simple characterizations of their states (e.g. their incoming or outgoing sequences). This motivates using quotient structures of automata to reason about abstract histories.

**Quotient Structure.** Given an equivalence relation $R$ on the states of a history automaton, the quotient structure of the automaton overapproximates the history by collapsing together equivalent states: Let $[q]$ denote the equivalence class of $q$. Then $\text{Quo}_R(h) = (\Sigma, \{[q] \mid q \in Q\}, \text{init}, \delta', \{[q] \mid q \in F\})$, where $\delta'([q], \sigma) = \{[q'] \mid \exists q'' \in [q] : q' \in \delta(q'', \sigma)\}$.

**k-Past Abstraction with Locking.**

API usage sequences often have the property that a certain sequence of events is always followed by the same behaviors. This motivates an equivalence relation in which states are considered equivalent if they have a common incoming sequence of length $k$ (e.g., [30]).

In order to differentiate between transitions in different locking contexts, we increase the length of the sequence to be considered for equivalence when the last events are lock events

**Definition 4.2.1** The $k$-past abstraction with locks is a quotient-based abstraction w.r.t. the $k$-past relation with locks $R[k]$ defined as: $(q_1, q_2) \in R[k]$
Figure 4.1: Example - states throughout a method analysis
if \( q_1, q_2 \) have the same locks acquired and have a common incoming sequence of \( k \) non-lock events

**Abstract Histories: Transformers.** In the concrete semantics, a concrete history is updated by either initializing it to a given history, or appending an event to it. The abstract semantics is defined via the following transformers:

- **Abstract extend transformer:** appends the new event \( \sigma \) to the final states, and constructs the quotient of the result.
- **Merge operator:** constructs the union of the given automata and returns the quotient of the result.

The abstract history component for a fresh object is initialized to a history reflecting an \texttt{init} event, preceded by lock events when needed. When an observable event occurs, the semantics updates the relevant histories using the extend transformer.

As long as the domain of abstract histories is bounded, the abstract analysis is guaranteed to terminate. Yet, in practice, it can easily suffer from an exponential blowup due to branching control flow. The merge operator mitigates this blowup, accelerating convergence. Specifically, at control flow join points, all abstract histories associated with the same abstract object (representing different execution paths) are merged. This introduces additional overapproximation but reduces the number of tracked histories.

**Example 4.2.2** Fig. 4.2 shows a subset of the (unbounded) concrete history set, which is merged into the abstract history, also shown in Fig. 2.3.
Figure 4.2: Example - abstract history creation
Chapter 5

Experimental Evaluation

5.1 Prototype Implementation

An interprocedural analysis was made using the WALA framework, in which the algorithm described in [25] was implemented. The analysis was tweaked in order to:

- The code analysis mostly relies on the framework’s may-point allocation site based analysis for object identification, but for the analyzed objects, it adds custom-made points-to analysis (described in [11] for more precision.)

- Start the analysis with every public method as an entry point.

- Avoid analyzing the API methods since they are the building blocks of the analysis. However in some methods of interest an artificial allocation site is created to enable analyzing the resulting object.

The analysis’s input is a compiled Java library. We used a special library crafted from multiple code samples using a specific concurrent data structure (e.g., ConcurrentHashMap). The compilation of the library may be done using a specialized partial compiler [9]. Receiving source files was done using ad-hoc scripts, since we found no API to crawl the code examples and obtain a sufficient number of code samples automatically.

Options which classes to analyze, which methods to ignore / analyze and other settings were implemented as well.
5.2 Results

5.2.1 Locking Patterns Study

In Tomcat 5.0 we observed noticeable number of accesses to data structures protected by a lock. The most common case was a global lock, such as in Fig. 5.3.

A special case has was a lock on single-method sequence. This case is interesting because if every lock sequence is a single-method one, the API can be immediately transformed to a concurrent data structure. There are simple single-method examples such as in Fig. 5.4, or sequences that can be easily transformed to single-method ones. For example, the sequence in Fig. 5.1 can be replaced by the single method sequence in Fig. 5.2.

Some sequences weren’t locked at all, because they were sequences on a local (non-shared) variable.

5.2.2 Concurrency Patterns (including locks)

Two experiments were conducted on objects of `java.util.concurrent.ConcurrentHashMap`. The first experiment was investigating 23 known concurrent libraries. Most of the libraries were taken from [27], and for them it was known they contained a variation of either Pattern 1 (Fig. 5.5) or Pattern 2 (Fig. 5.6).
Figure 5.3: Global Lock Example

get-put under a lock

H0

LOCK(x)

H4

-Ljava/util/HashMap: get()-locked(x)

H3

-Ljava/util/HashMap: put()-locked(x)

H3

UNLOCK(x)

H2

Figure 5.4: Single Method Lock Example

clustering result #67

H0

LOCK(0)

H1

-Ljava/util/HashMap: get()-locked (0)

H2

UNLOCK(0)

H3
```
T f1(k, ...) {
    T v = map.get(k);
    if (v == null) {
        v = ...  // compute v
        v2 = map.putIfAbsent(k,v);
        if (v2 != null) {
            v = v2;
        }
    }
}
```

Figure 5.5: Pattern 1 - Atomic write after read

```
T v = map.get(k);
if (v == null) {
    v = ...  // compute v
    map.putIfAbsent(k,v); // no null comparison of the returned value
}
```

Figure 5.6: Pattern 2 - Non-atomic write after read - no null check on putIfAbsent
Figure 5.7: Pattern3 - Non-atomic ”put if absent” - usually using Map interface
Figure 5.8: Pattern 4 - Locked write after read (or multiple writes)
1 T f5(k, ...) {
2     T v0 = map.get(k);
3     // process v0, may return
4     map.put(k, v); // write method
5 }

Figure 5.9: Pattern5 - Write after read without lock - except Pattern 3

1 T f6(k, ...) {
2     putIfAbsent(k, ...) // or 3-parameter replace (a single concurrent method)
3 }

source: patterns.MapPattern1.java/Map/putIfAbsent

Figure 5.10: Pattern6 - A single concurrent write
**Performance.** For a 6-core Intel Xeon CPU X5650 @ 2.67 GHz with 48GB RAM the run time of full analysis of the library, meaning using every public method as a pseudo entry point and track histories from `ConcurrentHashMap` objects (in most cases there were no such objects / histories) was 10 hours, when each entry point was given a thread out of 6-thread pool. Further runs of the analysis could take less than an hour if the "uninteresting" method list from previous run is provided.

The patterns in Fig. 5.10 - Fig. 5.7 describe other patterns emerging from these libraries with wide evidence there.
The following table describes the number of each one of the patterns detected by the tool in each benchmark:

<table>
<thead>
<tr>
<th>Benchmark project</th>
<th>#Java LOC</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
</tr>
</thead>
<tbody>
<tr>
<td>amf</td>
<td>4899</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Apache Cassandra</td>
<td>55483</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>beanlib 5.0.2</td>
<td>5512</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tomcat 7 (catalina.jar)</td>
<td>100133</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>Clojure 1.0</td>
<td>26911</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Apache Derby 10.8.2.2</td>
<td>258241</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DWR 3.0</td>
<td>35613</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>dynuproject-json</td>
<td>469</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ehcache-spring-annotations-1.2.0</td>
<td>3564</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>entity-fs 1.2</td>
<td>N/A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>geronimo-kernel-2.1.2</td>
<td>17074</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>graniteds-2.2.0.GA</td>
<td>26775</td>
<td>8</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>hazelcast-2.5</td>
<td>63941</td>
<td>6</td>
<td>0</td>
<td>60</td>
<td>3</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>hwajjin-1.0</td>
<td>4371</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ifw2</td>
<td>N/A</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jboss 2.1.8GA</td>
<td>N/A</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>jsefa</td>
<td>N/A</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>61</td>
</tr>
<tr>
<td>keyczar 0.7</td>
<td>775</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>myfaces 2.1.10</td>
<td>164481</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>openejb 4.5.1</td>
<td>128167</td>
<td>21</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>org-ektorp-1.1.1.jar</td>
<td>6154</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>resteasy-jaxrs-3.0-beta-3</td>
<td>33716</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>struts2-core-2.3.8</td>
<td>33651</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
T f i(k, . . . ) {  
T v = m a p . g e t ( k ) ;  
if ( v == null ) {  
v = . . . // compute v  
v 2 = m a p . p u t I f A b s e n t ( k , v ) ;  
v = m a p . g e t ( k ) ;  
}
}

Figure 5.11: Pattern 7 - Atomic write after read (a variation)

The second experiment was wider, and more focused to the concurrent data structure usage. At first, we obtained 10535 distinct Java files with java.util.concurrent.ConcurrentHashMap usage, which summed to a 1M lines of code. These files were obtained by download-friendly code search engines allowing source search and Java file download - Black Duck Open Hub Code Search [14] and Krugle Open Search [2]. Then, they were compiled using a partial compiler, since the full code and the dependencies were unavailable.

The resulting class files were added to a single JAR file, and the analysis was conducted on every public method of the resulting pseudo-library. The following table describes the patterns emerging from this analysis. P1-6 are the patterns described above, P7 is described in the Discussion section.

<table>
<thead>
<tr>
<th>#Java LOC</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
</tr>
</thead>
<tbody>
<tr>
<td>933609</td>
<td>99</td>
<td>51</td>
<td>108</td>
<td>129</td>
<td>581</td>
<td>88</td>
<td>30</td>
</tr>
</tbody>
</table>

5.2.3 Discussion

Comparing the pattern histogram in the 2 experiments, we found out that the pattern distribution is similar, providing evidence that the patterns identified above are the most common usage patterns in this specific case - the composition of ConcurrentHashMap atomic operations.

An additional pattern (counted in the P7 column of the second experiment) emerging from the analysis, which comes "between" Pattern 1 and Pattern 2, described in Fig. 5.11) was seen 30 times. This is a concrete code example of this pattern:
public void addRole(String user, String role) {
    Set<String> users = roles.get(role);
    if (users == null) {
        users = new CopyOnWriteArraySet<String>();
        roles.putIfAbsent(role, users);
        users = roles.get(role);
    }
    users.add(user);
}

Although the code is atomic as the code in the previous pattern, it differs:

- The calling thread doesn’t know that it is the one who added the object. The pattern doesn’t fit if actions should be made on an object in the case it was not added.
- The pattern contains an unnecessary composition of 3 methods when composition of only 2 methods is required.

**Pattern 2 instances - discussion:** (Fig. 5.6)

In 19 of 26 tested unique cases, manually studying the code revealed actual behavior - there is no indication which object was added to the map.

While it is usually a bug and this wrong value may be returned and used further, in other cases it is adding an object to the map, being aware of non-atomic behavior (noticing it in a code comment).

**Pattern 3 instances - discussion:** (Fig. 5.10)

This pattern, which isn’t atomic, is commonly used for ”compute if absent” for simple immutable computations, when the user does not care the computations are made in parallel and multiple times. In some cases, a comment in the code reveals the API user is aware of this behavior.

**Pattern 5 instances - discussion:** (Fig. 5.9)

This pattern can indicate a variety of bugs, but also can emerge from a correct code. Here are cases of this pattern found in the analyzed libraries:

Incorrect behavior:

- A straightforward atomicity violation - user may not write a value they intend to write - e.g. invalid ”compute if condition holds” (similar to ”compute if absent” as described earlier) - the computed value does not end up in the map and the user is not aware of it.
• Code smells (not atomicity violation but code not recommended to use) - such as the following method composition. In the following example, the get method is redundant and put only can be used.

```java
V currValue = map.get(key);
if (currValue != value) {
    valueAdded(value);
}
V oldValue = map.put(key, value);
if (oldValue != null) {
    valueRemoved(oldValue);
}
```

can be transformed to:

```java
V oldValue = map.put(key, value);
if (oldValue != value) {
    valueAdded(value);
    if (oldValue != null) {
        valueRemoved(oldValue);
    }
}
```

The second approach is better because it does not use unnecessary compositions.

Correct behavior:

• Valid "compute if condition holds", when the static analysis cannot decide whether the condition actually fits an atomicity violation or the correct usage.

• As it pattern 3, the user can sometimes be aware that computations are made in parallel and multiple times.

• Implementing locks which cannot be detected as Java monitors, (implementation of Lock interface etc.). A simple modification can be made to our analysis to catch these instances.

```java
Lock l = ...;
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```
• The sequence \texttt{remove after get} is atomic when 2-parameter \texttt{remove} is used.
Chapter 6

Related Work

*Checking Concurrency Operation Usage.* Part of our work and examples was inspired by the work of Shacham et al. [27] This work combines static and dynamic analyses to test concurrent composed operations for atomicity. Their another work [28] proves atomicity efficiently. Our approach is applying static analysis only and using also partial code examples.

Lin and Dig [16] implemented a tool named CTADetector detecting API misuses in concurrent data structures. Their approach checks the code for patterns resembling the patterns in our work, they use “Check and act” notation for concurrent data structure usage. Their tool successfully detects *priorly known* patterns in user’s code and proposes fixes for patterns representing misuse. They do not use typestate abstractions for the patterns and do not learn the patterns themselves as done in our work. Our work is also more general, uses point-to analysis and native interprocedural support.

Befrouei et al. [6] present concurrency pattern detection in execution traces used to detect data race patterns there.

Kumar [15] presents dynamic specification mining of DFA on the level of messages between components in a distributed system.

Pradel et al. [24] present a technique extracting typestate DFAs (as described in [23]) from dynamic analyses and translating them into relationship-based constraints, and then using static checker for anomaly detection. One of the examples there is the lock usage specification in a multi-threaded environment. Our approach avoids using dynamic analysis and mines the specifications statically, it also applies interprocedural analysis absent from
the work.

**Static Specification Mining - Without Concurrency.**

Shoham et al. [30] use a whole-program analysis to statically analyze clients using a library and extract temporal specification DFAs. Since they always employ a whole-program analysis, their approach is not applicable in our setting, because our work was based on a library itself or on a single source file. Furthermore, their approach is limited to single-object typestate.

The work of Mishne et al. [19] performs analysis and extracts temporal specification DFAs on single source files only. Their work is focused on code search and completing missing data, not on precise specification mining needed for synchronization pattern detection. Their approach for unknowns detection was generalized and formalized in the work of Peleg et al. [22]

Specification mining for other, not DFA-based types were described in the latest years. The work of Stolee et al. [31] present a technique of extracting usage examples from a given specification. Sanitar et al. [26] present specification mining from math related APIs using machine learning.

**Dynamic Specification Mining - Without Concurrency.** Dynamic analysis does not suffer from the difficulties inherent to abstraction required in static analysis. Thus, when it is feasible to run a program with adequate coverage, dynamic analysis represents an attractive option for specification mining.

Cook and Wolf [7] consider the general problem of extracting an FSM model from an event trace, and reduce the problem to the well-known grammar inference [13] problem. Cook and Wolf discuss algorithmic, statistical, and hybrid approaches, and present an excellent overview of the approaches and fundamental challenges. This work considers mining automata from uninterpreted event traces, attaching no semantic meaning to events.

Ammons et al. [4] infer temporal and data dependence specifications based on dynamic trace data. This work applies sophisticated probabilistic learning techniques to boil down traces to collections of finite automata which characterize the behavior. Lo and Khoo [17] extend Ammons’ work, and employ machine learning techniques in order to mine probabilistic temporal specifications from dynamic execution traces.

Whaley et al. [35] present a two-phased approach to mining temporal API interfaces, combining a static component-side safety check with a dynamic client-side sequence mining. The static analysis is used primarily to
restrict the dynamic search of temporal sequences, rather than to directly infer specifications.

Perracotta [36] addressed challenges in scaling dynamic mining of temporal properties to large code bases, and focuses on mining traces for two-event patterns with an efficient algorithm.

Dallmeier et. al. [10] use a dynamic analysis to extract object behavior models (similar to temporal specifications) from executions. Their approach uses a preceding static analysis that classifies each method as a mutator (modifies object state) or an inspector. Mutator methods are instrumented to record the resulting object state.

Component-side Static Analysis. In component-side static analysis, a tool analyzes a component’s implementation, and infers a specification that ensures the component does not fail in some predetermined way, such as by raising an exception. In contrast, client-side mining produces a specification that represents the usage scenarios in a given code-base. The two approaches are complementary, as demonstrated in [35].

Alur et al. [3] use Angluin’s algorithm and a model-checking procedure to learn a permissive interface of a given component.

Gowri et al. [20] use static mining to learn a permissive first-order specification involving objects, their relationships, and their internal states.

Client-side Static Analysis. A few papers have applied static analysis to client-side specification mining.

Weimer and Necula [34] use a simple, lightweight static analysis to infer simple specifications from a given codebase. Their insight is to use exceptional program paths as negative examples for correct API usage. We believe that our approach could also benefit from using exceptional paths as negative examples. Weimer and Necula learn specifications that consist of pairs of events \( \langle a, b \rangle \), where \( a \) and \( b \) are method calls, and do not consider larger automata. They rely on type-based alias analysis, and so their techniques should be much less precise than ours. On the other hand, their paper demonstrates that even simple techniques can be surprisingly effective in finding bugs.

Mandelin et al. [18] use static analysis to infer a sequence of code (jungloid) that shows the programmer how to obtain a desired target type from a given source type. This code-sequence is only checked for type-safety and does not address the finer notion of typestate. Thummalapenta and Xie [32]
introduce a tool called PARSEWeb to expand on Mandelin’s approach by gathering samples online, partially-compiling them and analyzing the results with a simple static analyzer. We employ a similar technique in the first phases of our solution, and we draw from their experience. Like with Mandelin’s work, however, their analysis is only concerned with the object types appearing in code sequences.

Wasylkowski, Zeller, and Lindig [33] use an intraprocedural static analysis to automatically mine object usage patterns and identify usage anomalies. Their approach is based on identifying usage patterns. In contrast, our preliminary work mines temporal specifications that over-approximate the usage scenarios in a code-base, which is a feature we intend to maintain in future work.
Chapter 7

Conclusion

We present a framework for static mining of synchronization patterns from “big code”. Our approach is based on static analysis of a large number of client programs that use concurrent objects. By statically analyzing each client program, we extract the way in which it uses the concurrent object and represent it semantically using finite-state automata. The extracted automata tracks both the method invocations performed by a client on the concurrent object, and the synchronization operations (if any) performed by the client. The automata extracted from each individual client program are later consolidated into a common representation, such that common patterns can be identified. Our approach handles both lock-based synchronization and lock-free patterns. We have implemented our approach in a tool called SynchMiner based on the WALA static analysis framework, and applied it to identify 7 major patterns by analyzing 23 real-world concurrent full libraries and also different 10000 class files with concurrency.
Bibliography


The experiments were conducted to analyze synchronization patterns in a concurrent hash map, the java.util.concurrent.ConcurrentHashMap.

The first experiment examined the use of the data structures in 23 popular concurrent libraries. Most of these libraries were chosen since it was known beforehand that each library contained or was likely to contain an instance of the "Compute and Increment if Not Exist" pattern.

In the second experiment, we developed a partial example of using the data structure in the form of a few Java files. These files were downloaded using search engines. We downloaded 10,535 files, of which 30 of them contained the data structure, totaling a million lines of code, and we created a binary library using the compiler.

The results of the experiments are presented in Section 5.2. We found that the distribution of patterns between the two experiments was similar, which supported our hypothesis that these are the most common patterns used in the data structure.

The pattern "Compute and Increment if Not Exist" was actually seen approximately 54 times, of which 26 were manually corrected and 26 were found to be functional errors due to the incorrect use of the pattern.
מטעלי גלובלי – רף הקיריאוט נוספים חתות מנעול.

• מטעלי גלובלי – רף שאורה יニック באמצעות ברקאייה בסיס. עוניני ברצפים מצטברعام.

• שנית להכרת את הרצף המלוכל בפעולה בודדים על פניהם מקבילים.

• "Mission Tips" – קיראה לвлажн עילית שהיחידה ייש מחשב. דפוס זה משני כריאה.

• רף קיריאוט לﻸל עילית – רף שאורה יニック אפיי MORE מקבילים עד רף אחר על מבנה.

הנוטוים

**דפוסי סנכרון על מבני נתונים מקבילים**

דפוס מחוכלית זה היה רף קיריאוט על מבנים נתונים מקבילים, המבטאת תכונה מסויימת כנף.

אות輯יות. הנוגעות לדפוסים אלה מוצגים בתיעודים 2.1-2.3.

* פעולות על פניהם מתווגים המורכבים העתקות סמלים

* "חשב והכנס אם לא קיים בקומפ אוטומט" – ללא מנעולים.

* דפוס המישל כילה – "חשב עד התחלקה".

**הדפים המופרים שלולית חיונ:

• "חשב והכנס אם לא קיים בקופ יחשוב" – ללא מנעולים – תרשים 5.5.
• "חשב והכנס אם לא קיים בקופ יחשוב" – ללא מנעולים – תרשים 5.6.
• "חשב והכנס אם לא קיים בקופ יחשוב" – ללא מנעולים – תרשים 5.7.
• רף "kahob חשה כייריאום" החת מתוכלע – תרשים 5.8. בדות לכל, התוונה היא לכל לחוב אס.
• רף "kahob חשה כייריאום" – ללא מנעולים – תרשים 5.9.
• פעולות מוכבדת בודדות על מבנה נתונים מקבילים.

**🌙 ניסיונים אמפיריים

הניסיון הון הרוצת של כלל מיפור. מרכבי כלל העיקריות חונים:

• נגזרת בקוחית אוטומט מצבי-ıtוכן-טיפס מועשר במעברש "הידידי למרכיבי סנטור" / מובילים.
• סוכנו ימייה את אילולים ריבוע השבט לש מטדור, שבכוכדות השבט מבחר את האוטומטוס "יהל אילבוקט". החטואים למטדור עניי החטואיות הIBUTES ורמות ושיחמות WALA. בהל מתודת המופית קונבוצות גישה לתוכנות ע未來ית יונית פימי של חקיאום.
• המוכרבים את מכבר האוטומטוס.

שני ניסיונים לברירת דפוסים סנסור על מבני נתונים מקבילים
מכיוון שלא מתבצע ניודון מלא של תוכנית, אוטומט של אובייקט לא לכלול בדר כל התוכנית. ייצורו, אלא י섬ק את רפים האטרזים עליה מחכיתת התוכנית ועדיין.

האוטומטים משקפים פעולות על אובייקטים ממตกแต่งות בדד, שווה האובייקטحت לאובייקט, ועילה / שחרור חשוב.

şıוואה של רכז החזרה מספר אפי פועלות על אובייקטים, ועילה.

 emploi הקוריר וחרת על כל האובייקטים.

 emploi הבקבוקים על אסまず

 이것이 רץ הקריאות על הסבר באורכי על אובייקטים. החנה מ단체ים בבנייה האוטומטים.

 אני מצפים את האובייקטים על התוכן. כי אובייקטים מתאימים / מפר中介机构 כל מכובה מסולוגיה

 הנינשה אולית. מסלול הנישה מרכזים הקריאות (ה눔יגה מרשימת האטרזית - פרצדרול) מוזה של האובייקטים בתוכו המחלקה, הנקבע על ייעוד ה"פצל" לצבועי.

 WALA

 לידך האוטומטים

 אחר מהתוכנים ברסמ שיווק אוטומטים, המתחברות כדלקבון מקבץ דפים שונים. 

 • יותח – על אוטומטים זהים או המעבדים גוזים עד כדמוי אובייקטים מבער

 • הנעילה משושים לאות המكهرب.

 • סמנטיות הקריאות – של אוטומטים חיים לאות מרגרפ ואס האוטומטים המתחבולים

 • אחריו מתוכדו חביבה במעודה סמנטיות שול – שיכולה להתקיים מספר קריאות

 • שוחט – זויס לפי הגזרת ההוזה עליל.

 • התכל – אוטומטים המכל אוטומטים אחר, דהינו מתכבול מעניין הסבר מעצבי וא

 • מعقبים, נוכס למקבץ של. השיתוף בינוכץ כו באורח הלוך לבבל, פי עצים

 • שימשו את הכור מציאתי נוכזו, פי שרואים בחרישים 5.6.5.5, צירוף למקבץ אוחז

 • פוטוידוע עם דפוסים דומים היכולים קריאות על פי התוכן התכלית או אחר מצט סופי

 "בור".

 סוג דפוסים minecraft

 דפוס סעילה

 דפוס סטנילי ניתן למצוא את תיאור התוכן תעשיה. סוג מדעות ברעון, אני הכל

 מטשו хотитеblems כדלקבון:

 • מחלקות מ-ln

 • בולם מ-נ

 • משטרני תאני

 java.util.concurrent.lock.Locks
We present a framework for static synchronization of patterns from large code repositories.

The basic idea is to use static analysis to produce synchronization patterns from a large number of applications that use parallel components and to summarize the produced patterns to uncover common patterns in the codebase.

The foundation of our analysis is an abstraction based on automata that captures reads and synchronization operations of the code. By combining reads with synchronization operations, we reveal common synchronization patterns in the basis.

Caching is focused both on memory and non-blocking.

We implement our approach using the SyncMiner tool based on the WALA framework and applied it to 23 common libraries worldwide that use the parallelism, and in addition from 10,000 code repositories using synchronization techniques. Figures 2.7 illustrate SyncMiner's operation on binary code libraries, both real and synthetic libraries created by partial compilation of example code from search engines.

The tool analyzes each public method as an entrance point to the code and analyzes all objects under analysis for each entrance method. We achieve the temporal characteristics – the tool analyzes binary code libraries and tracks each method that is called from each object.

We analyze the return values of methods "interesting" in order to separate synchronization patterns.

Technion - Computer Science Department - M.Sc. Thesis 2015
המחקר בוצע להנחייתו של פרופסור ערן יהב, מהפקולטה למדעי המחשב.

אני מודה לטכניון על התמיכה הכספית הנדיבות, מחוקולוגיה למודי המחשב.

אני מודה לטכניון על התמיכה הכספית הנדיבות, מחוקולוגיה למודי המחשב.

אני מודה לטכניון על התמיכה הכספית הנדיבות, מחוקולוגיה למודי המחשב.

אני מודה לטכניון על התמיכה הכספית הנדיבות, מחוקולוגיה למודי המחשב.

אני מודה לטכניון על התמיכה הכספית הנדיבות, מחוקולוגיה למודי המחשב.
כריה סטטית של דפוסי מקביליות נפוצים

חיבור על מחקר

לשם مليוי החלק של הדרישות לשימושי קבליות תורתיות מגיסטר למדעי מחשב

יבגני אברמוביץ'

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

ניסן התשע’יות

מרץ 2015
כריה סטטית של דפוסי מקבליות נפוצים

יבגני אברמוביץ‘