Sketching Concurrent Data Structures

Armando Solar-Lezama, Christopher Grant Jones, Rastislav Bodík
University of California, Berkeley
{asolar,cgjones,bodik}@eecs.berkeley.edu

Abstract

We describe PSKETCH, a program synthesizer that helps programmers implement concurrent data structures. The system is based on the concept of sketching, a form of synthesis that allows programmers to express their insight about an implementation as a partial program: a sketch. The synthesizer automatically completes the sketch to produce an implementation that matches a given correctness criteria.

PSKETCH is based on a new counterexample-guided inductive synthesis algorithm (CEGIS) that generalizes the original sketch synthesis algorithm from [20] to cope efficiently with concurrent programs. The new algorithm produces a correct implementation by iteratively generating candidate implementations, running them through a verifier, and if they fail, learning from the counterexample traces to produce a better candidate; converging to a solution in a handful of iterations.

PSKETCH also extends SKETCH with higher-level sketching constructs that allow the programmer to express her insight as a "soup" of ingredients from which complicated code fragments must be assembled. Such sketches can be viewed as syntactic descriptions of huge spaces of candidate programs (over 10^8 candidates for some sketches we resolved).

We have used the PSKETCH system to implement several classes of concurrent data structures, including lock-free queues and concurrent sets with fine-grained locking. We have also sketched some other concurrent objects including a sense-reversing barrier and a protocol for the dining philosophers problem; all these sketches resolved in under an hour.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.2 [*Software Engineering*]: Design Tools and Techniques

General Terms Languages, Design

Keywords Sketching, Synthesis, Concurrency, SAT, SPIN

1. Introduction

Data structures designed to be shared among many concurrent threads are among the most complex programs one can write in less than a thousand lines of code. The source of this complexity can be traced back to the requirement that the data structure maintain consistency in the presence of many simultaneous updates. Moreover, programmers must maintain this consistency while keeping mutual exclusion to a minimum, in order to prevent the data structure from becoming a sequential bottleneck in a highly concurrent application. In order to achieve this, data-structure designers must resort to complex schemes to maintain the consistency with only fine-grained locking, or even without using locks at all, relying only on atomic primitives provided by the hardware. Finally, the composition of concurrent objects is far from trivial, so library-based approaches will not shield programmers from the complexities of concurrent data structures.

In this paper, we argue that program synthesis in the form of sketching can be an important element in helping programmers cope with these daunting challenges. Sketching is a form of software synthesis designed to make programming easier by helping programmers focus on the high-level implementation strategy and leave the low-level details to the synthesizer. With sketching, the programmer creates an implementation by writing a sketch — a partial program containing only the easier-to-write parts of the code, together with additional insight to help synthesize the remaining "holes". In this way, the programmer is relieved from the most demanding aspects of programming, while still maintaining full control over the implementation.

Previous work applied Sketching to the development of ciphers and error correction codes, and to important classes of scientific programs [18, 20]. But the PSKETCH synthesizer described in this paper is the first sketch synthesizer capable of reasoning about concurrency. PSKETCH extends the original SKETCH system with a new synthesis algorithm based on the concept of counterexample-guided inductive synthesis (CEGIS). The new system also adds high-level sketching constructs to the original language, making it easier to express insights about the implementation without having to think about the details.

Like its predecessor, the PSKETCH synthesizer performs combinatorial synthesis, framing the synthesis problem as a search for a sketch completion that satisfies a given correctness criteria. The synthesis algorithm uses counterexample-guided inductive synthesis (CEGIS) to search this space efficiently. CEGIS works by selecting candidate implementations from the space, then using the counterexample produced by a verification procedure to prune a large fraction of the search space when the selected candidate is shown to be wrong. The key innovation in PSKETCH is to reformulate CEGIS for the case when the counterexample produced by the verifier is no longer an input, but an execution trace on the selected candidate. By reformulating CEGIS in this way, we are able to take any verifier capable of producing counterexample traces and use it to build a sketch synthesizer.

The new high-level sketching constructs make it easy for the programmer to use the synthesizer. For example, the programmer can now ask the synthesizer to discover the correct ordering of operations in a block. This is especially useful in the concurrent setting, where programmers must often expend considerable effort determining the right point to release a lock, or the right way to order a sequence of updates to shared data. The new constructs also make it simple to constrain the set of pointer expressions that the synthesizer can use to complete a pointer-valued hole. This makes it very easy for programmers to provide partial insights about complex pointer-manipulations.

We have used the PSKETCH system to implement several classes of concurrent data structures, including lock-free queues and concurrent sets with fine-grained and optimistic locking . We have also sketched some interesting concurrent objects including a sense-reversing barrier and a protocol for the dining philosophers problem. The synthesizer is able to quickly search through enormous spaces of candidate programs; in one of our benchmarks, for example, the synthesizer was able to find a correct implementation for a lock-free queue from a space of more than 10^8 candidate implementations in about 50 minutes.

In summary, the key contributions of the paper are.

1. A generalization of the CEGIS approach to synthesize concurrent programs.
2. A set of language extensions and high-level constructs to support sketching of concurrent data structures.
3. An experimental evaluation of sketching for concurrent data structures.

Section 2 is a tutorial on sketching for concurrent data structures. Section 3 provides a brief introduction to the sequential SKETCH language, and Section 4 introduces the extended PSKETCH language. Section 5 describes the CEGIS algorithm for sequential programs, and Section 6 shows how we generalized it to handle concurrent programs. Section 7 demonstrates how the new sketching constructs are implemented on top of the base language, and Section 8 contains our empirical evaluation of PSKETCH.

2. Sketching with Concurrency

In this section we introduce sketching of concurrent data structures from the programmer's point of view. We show how—with only a few constructs for sketching concurrent operations—the PSKETCH language allows the programmer to express enough of the structure to synthesize a correct and efficient implementation, all the while having only a partial knowledge about how the final program will work. We will walk through the development using a problem assigned two years ago in a undergraduate exam on operating systems.

We start by quoting the exam problem. Deceitfully simple, the problem was successfully answered by less than 30% of the students, even with additional hints (which we omitted).

Lock-Free Queue. An object such as a queue is considered "lock-free" if multiple processes can operate on this object simultaneously without requiring the use of locks, busy-waiting, or sleeping. We will construct a lock-free FIFO queue using an atomic "swap" operation. This queue needs both an `Enqueue` and a `Dequeue` method.

Instead of the traditional `Head` and `Tail` pointers, we will have `PrevHead` and `Tail` pointers. `PrevHead` will point at the last object returned from the queue, so `PrevHead.next` will point to the head of the queue. Here are the basic class definitions, under the assumption that only one thread accesses the queue at a time.

```
// Holding cell for an entry
class QueueEntry {
  QueueEntry next = null;
  Object stored;
  int taken = 0;
  QueueEntry(Object newobject) { stored = newobject; }
}
```

```
// The actual Queue (not yet lock-free!)
class Queue {
  QueueEntry prevHead = new QueueEntry(null);
  QueueEntry tail = prevHead;
  void Enqueue(Object newobject) {
    QueueEntry newEntry = new QueueEntry(newobject);
    tail.next = newEntry;
    tail = newEntry;
  }
  Object Dequeue() {
    QueueEntry nextEntry = prevHead.next;
    while (nextEntry != null && nextEntry.taken == 1)
      nextEntry = nextEntry.next;
    if (nextEntry == null)
      return null;
    else {
      nextEntry.taken = 1;
      prevHead = nextEntry;
      return nextEntry.stored;
} } }
```

Suppose that we have an atomic swap instruction that takes a local variable (register) and a memory location and swaps their contents. In a relaxed dialect of Java that allows pointers, it can be described as follows.

```
Object AtomicSwap(variable addr, Object newValue) {
  Object result = *addr;  // Get old value (object)
  *addr = newValue;       // Store new object
  return result;          // Return old contents
}
```

Problem (a). Using the `AtomicSwap()` operation, rewrite code for `Enqueue()` such that it will work for any number of simultaneous `Enqueue` and `Dequeue` operations. You should never need to busy wait. Do not use locking (*e.g.*, test-and-set lock). Although tricky, it can be done in a few lines.
Problem (b). Rewrite code for `Dequeue()` such that it will work for any number of simultaneous threads working at once. Again, do not use locking. You should never need to busy-wait. □

This problem gives away more about the final solution than sketching typically requires, yet it is interesting enough to illustrate how sketching is helpful. The following development reflects the actual sketching process by a co-author who had not previously seen the solution to this problem.

Let us first consider how the programmer might sketch the concurrent `Enqueue` operation. First, the programmer speculates that, in addition to the initialization of a new entry, the method will consist of one or more of following statements:

assignment	::=	location = value
swap	::=	`tmp = AtomicSwap`(location, value)

The next step is to come up with locations and values that the concurrent `Enqueue` may need to reference. The programmer guesses that these sets are sufficient overestimates:

location	::=	{`tail`, `tail.next`, `newEntry.next`, `tmp.next`}
value	::=	location ∪ {`tmp`, `newEntry`, **`null`**}

Next, the programmer realizes an important implication of the `AtomicSwap` semantics. Unlike CAS, whose typical use is to update the data structure only when a race condition has not occurred, `AtomicSwap` modifies the location unconditionally. Therefore, if the swap fails, some corrective action may be necessary.

The programmer of course does not know what it means for the swap to fail, or whether it can fail at all, because he does not know the solution. He can, however, state his observation by adding

```
#define aLocation {| tail(.next)? | (tmp|newEntry).next |}
#define aValue    {| (tail|tmp|newEntry)(.next)? | null |}
#define anExpr(x,y)  {| x==y | x!=y | false |}

void Enqueue(Object newobject) {
  QueueEntry tmp = null;
  QueueEntry newEntry = new QueueEntry(newobject);

  reorder {
    aLocation = aValue;
    tmp = AtomicSwap(aLocation, aValue);
    if (anExpr(tmp, aValue)) aLocation = aValue;
  }
}
```

Figure 1. A sketch for the concurrent Enqueue.

a fixup statement to the set of statements that may comprise the concurrent Enqueue.

fixup	::=	**if** (expr (tmp, value)) assignment
expr(x,y)	::=	x == y | x != y

At this point, the programmer believes to have listed a superset of the statements that the concurrent Enqueue might need. He does not know how to assemble these statements into a working method, but he can already express an *informal sketch* of the desired Enqueue method:

> The concurrent Enqueue method will execute—in *some sequential order*—(i) an assignment, (ii) a swap, and (iii) an optional fixup statement.

This is the extent of the reasoning that needs to be carried out about the concurrent Enqueue, and the resulting informal sketch is all that the synthesizer needs to know to perform the synthesis. We are now ready to write the sketch in the PSKETCH language. The sketched Enqueue is shown in Figure 1. Since it closely corresponds to the informal sketch, little explanation is in order. First, note that macro definitions behave as in C. Second, the PSKETCH expression **{|** e_1 | e_2 | . . . **|}** asks the synthesizer to select one of the e_i expressions, which can be given as regular expressions. Third, the **reorder** construct specifies that the statements in its body can appear in any order in the final implementation. Finally, note that the programmer included among the choices for anExpr the **false** expression; this makes the fixup statement optional.

The value of the sketch for the programmer is highlighted by the fact that the sketched Enqueue represents 1,975,680 unique candidate programs. Since the synthesizer will select a correct one from among them, the programmer can now think in terms of coming up with a set of ingredients rather than how to orchestrate them. It remains to specify the correctness condition, so that the synthesizer can select a correct candidate. We discuss how this is done in Section 4, but here it suffices to say that the programmer-specified conditions require that the concurrent Enqueue obeys the same structural properties as the sequential counterpart that was given in the problem statement, and is sequentially consistent.

The resolution of the sketch in Figure 1 by the synthesizer produces the concurrent Enqueue method in Figure 2. The fixup statement was optimized away because it was unnecessary (the synthesizer replaced anExp with **false**).

The sketch for the concurrent Dequeue is shown in Figure 3. In this operation, the programmer easily realized that the main trick is to test the taken field with atomicSwap, so this aspect was not sketched. The tricky part was to come up with correct code that can advance the prevHead pointer as far as possible, for improved performance. This loop was sketched. In it, there is a choice of where to start the iteration (there were only two plausible choices) and a choice where to end the loop (again, only two choices). The sketch for Dequeue in Figure 3 represents only 4 programs, but together with the Enqueue sketch, the programmer has encoded over five million candidate implementations. Section 8 describes a sketch for Dequeue that corresponds to more candidates; that sketch seeks to synthesize a program that updates the prevHead pointer during the first loop. Such a Dequeue has incomparable performance with that in Figure 4 (depending on the workload, one or the other may be preferred). These two sketches give hope that sketches may be used to quickly develop alternative algorithms.

```
void Enqueue(Object newobject) {
  QueueEntry tmp = null;
  QueueEntry newEntry = new QueueEntry(newobject);

  tmp = AtomicSwap(tail, newEntry);
  tmp.next = newEntry;
}
```

Figure 2. The sketch from Figure 1, resolved.

```
Object Dequeue() {
  QueueEntry nextEntry = prevHead.next;
  while (nextEntry!=null &&
         AtomicSwap(nextEntry.taken,1)==1)
     nextEntry = nextEntry.next;
  if (nextEntry == null)
     return null;
  else {
     QueueEntry p = {| prevHead | nextEntry |};
     while (p != NULL && {| p(.next)?.taken |} ) {
       prevHead = p;
       p = p.next;
     }
     return nextEntry.stored;
} } }
```

Figure 3. A sketch for the concurrent Dequeue.

```
Object Dequeue() {
  QueueEntry nextEntry = prevHead.next;
  while (nextEntry!=null &&
         atomicSwap(nextEntry.taken, 1)==1)
     nextEntry = nextEntry.next;
  if (nextEntry == null)
     return null;
  else {
     QueueEntry p = prevHead;
     while (p != NULL && p.taken) {
       prevHead = p;
       p = p.next
     }
     return nextEntry.stored;
} } }
```

Figure 4. The sketch from Figure 3, resolved.

3. The SKETCH Language

To give necessary background for the sections that follow, we summarize here the SKETCH language introduced in [21]. This language supports sketching-based synthesis by extending a simple imperative language with a single synthesis operator on top of which higher-level and domain-specific synthesis constructs can be added as mere syntactic sugar.

With the SKETCH language, the programmer first writes a clean, behavioral specification for an algorithm, and then he sketches an outline of an efficient implementation. We have observed that this outline, called a sketch, captures the programmer's insight about the implementation while allowing the programmer to leave tedious details unspecified.

Let us illustrate programming with SKETCH using a small program submitted to a SKETCH programming contest that we organized in the past year. The contestant used SKETCH to implement a problem that he had previously solved by hand; this manual process took half a day. As we will see shortly, sketching the same implementation is much easier.

The problem at hand is to compute a 4×4 matrix transpose. The specification is given in the function `trans`. (Note that `trans` is an executable specification, not a declarative one, and so one can debug it easily with standard debugging techniques.)

```
int[16] trans(int[16] M) {
  int[16] T = 0;
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
      T[4 * i + j] = M[4 * j + i];
  return T;
}
```

While optimizing the transpose, the student realized that it might be possible to parallelize the transpose with a SIMD instruction called `shufps`. This instruction accepts two 4-word arrays and semi-permutes each into a 2-word array; the semi-permutations are given by a third argument. The following SKETCH function emulates the semantics of `shufps` in the SKETCH language.The indexing notation `a[b::c]` translates to a sub-array of `c` cells of array `a` starting at index `b`.

```
int[4] shufps(int[4] x1, int[4] x2, bit[8] b) {
  int[4] s;
  s[0] = x1[(int) b[0::2]];  s[1] = x1[(int) b[2::2]];
  s[2] = x2[(int) b[4::2]];  s[3] = x2[(int) b[6::2]];
  return s;
}
```

The student's insight was that a `shufps`-based transpose had to proceed in two stages: The input matrix had to be permuted into an intermediate matrix, which would then be permuted into the resulting (transposed) matrix. It was not immediately obvious, however, how exactly the two stages were to proceed.

The sketch `trans_sse` shown below expresses the student's insight. First, we need to introduce the sketch constructs in the language. The **`implements`** directive in the function header tells the synthesizer to resolve the sketch `trans_sse` such that it is behaviorally equivalent to `trans`, *i.e.*, the two must compute the same function. The **`??`** operators, called the *primitive hole*, will be replaced by the synthesizer with suitable constants to satisfy the behavioral equivalence. Finally, the `repeat(n) s` construct is a synthesis-time macro that `n` times replicates `s`. The replication creates fresh holes, each of may be replaced with a different constant.

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;
  repeat (??) S[??::4] = shufps(M[??::4], M[??::4], ??);
  repeat (??) T[??::4] = shufps(S[??::4], S[??::4], ??);
  return T;
}
```

The sketch concisely expresses the insight. Notice that the programmer fixed the two permutation stages but he left unspecified (1) the number of `shufps` instructions necessary for the task, (2) the ranges of matrix cells to be permuted, and (3) the bit vectors directing the permutations. The above sketch resolves in 33 minutes on a 1.G GHz Core 2 laptop. The synthesized `trans_sse` is shown below. (The binary strings are initializers for the bit-arrays, and are read left-to-right):

```
S[4::4]  = shufps(M[6::4],  M[2::4],  "11001000");
S[0::4]  = shufps(M[11::4], M[6::4],  "10010110");
S[12::4] = shufps(M[0::4],  M[2::4],  "10001101");
S[8::4]  = shufps(M[8::4],  M[12::4], "11010111");
T[4::4]  = shufps(S[11::4], S[1::4],  "10111100");
T[12::4] = shufps(S[3::4],  S[8::4],  "11000011");
T[8::4]  = shufps(S[4::4],  S[9::4],  "11100010");
T[0::4]  = shufps(S[12::4], S[0::4],  "10110100");
```

4. The Concurrent PSketch Language

The PSKETCH language extends the SKETCH language introduced in [20] in two important directions. First, it provides higher level sketching constructs with which programmers can more easily express their insights. Second, it introduces threads and synchronization primitives.

Concurrency introduces nondeterminism, which precludes the SKETCH approach of specifying a sketch's behavior by a reference implementation to which a resolved sketch must be functionally equivalent. At the end of this section, we describe how correct behavior is specified in PSKETCH.

4.1 High-level sketching constructs

The sequential SKETCH language extends its imperative base with a single synthesis construct: the primitive "hole" expression, **`??`**, which the synthesizer replaces with a constant that makes the sketch satisfy its specification. Prior work found the primitive hole sufficient for synthesizing expressions (r-values), such as loop bounds and index expressions in matrix manipulations. [20]

When sketching concurrent data structures, we found a need to synthesize (1) left-hand-side expressions (l-values) and (2) control flow, such as the order in which statements execute. To facilitate sketching of these constructs, PSKETCH introduces two features (i) regular-expression expression generators, (ii) a **`reorder`** block. As we will show in section Section 7, these constructs can be implemented as syntactic sugar on top of the basic **`??`** expression.

Regular-expression expression generators. Regular-expression generators (hereafter RE-generators) allow the programmer to sketch both r-value and l-value expressions from a restricted regular grammar.

The RE-generator construct has the form **`{|`**e**`|}`**, where e is a regular expression literal. The semantics of the construct is that the synthesizer substitutes the syntactic occurrence of the construct with a string from $L(e)$ such that the substitution makes the sketch satisfy its specification. RE-generator are not simply expanded as a macro, however; for programmability, we require that each component regular expression e be well typed.

RE-generators are typically used to enumerate symbolic memory locations or values that the synthesized code can reference. For example, the following PSKETCH fragment shows how we sketched the use of a *compare-and-swap (CAS)* instruction in a doubly-linked data structure.

```
#define NODE {| (tprev|cur|prev)(.next)? |}
#define COMP {| (!)? ((null|cur|prev)(.next)? ==
                      (null|cur|prev)(.next)?) |}
while(cur.key < key){
  Node tprev = prev;
  reorder {
    if (COMP) { lock (NODE); }
    if (COMP) { unlock (NODE); }
    prev = cur;
    cur = cur.next;
  }
}
```

Figure 5. A sketch of hand-over-hand locking.

```
while(cur.key < key){
    if (prev != null)
      unlock (prev);
    lock (cur.next);
    prev = cur;
    cur = cur.next;
}
```

Figure 6. The sketch from Figure 5, resolved.

```
CAS({| head(.next|.prev)? |},
    {| newNode(.next|.prev)? |},
    {| newNode(.next|.prev)? |})
```

The first CAS argument selects the location to be modified, and the second and third arguments give the old and new values, respectively. When writing the sketch, the programmer suspected (or insisted) that a CAS had to be used in the synthesized code, but he did not know which location had to be updated, and with what values. With the sketch below, he effectively specified all 27 CAS fragments that made sense in the context of the list addition operation (accessing other locations does not make sense in this operation).

RE-generators support only two regular expression operators: $e_1 \,|\, e_2$ and optional expressions $e?$.

At first sight, the exclusion of Kleene closure might seem arbitrary, but keep in mind that RE-generators are used to generate bounded program text. In real code, it is unusual to find chains of pointer dereferences of the form `{|p(.next)*|}` with more than two or three levels of dereferencing, so adding Kleene closure would increase the search space without any significant programmability benefit.

Reorder block. Concurrent data structures often depend on careful statement ordering to satisfy desired invariants. For this reason, we extended PSKETCH with a **reorder** construct that leaves the synthesizer in charge of determining the correct order for the statements in a block of code. The synthesizer considers all possible orders of these statements and selects one that, together with other choices made by the synthesizer, turns the sketch into a program that meets the specification.

In Section 2, we showed a sketch that used **reorder** to let the synthesizer decide where in a block of code to use an atomic swap. In many other sketches, we have similarly used the **reorder** block to describe a "soup" of operations which, when ordered in the right sequence, can produce a correct program.

Another use of **reorder** is to control mutual exclusion. For example, one of our benchmarks implements a hand-over-hand locking scheme for adding and removing elements from a concurrent set represented as a sorted linked list (see Section 8). As the algorithm scans the list, it must acquire and release some locks to maintain a sliding window of locks around the pointers it is holding. This scheme is tricky to get right, but we can use the **reorder** block to give the synthesizer the freedom to discover the correct strategy for acquiring and releasing the locks. The sketch is shown in Figure 5, and the synthesized code is shown in Figure 6. Note how the synthesizer used the freedom to reorder statements to discover the correct strategy for acquiring and releasing the locks. [1]

```
struct Lock {
   int owner = -1;
}                               unlock(Lock lk) {
lock(Lock lk) {                   assert lk.owner == pid;
  atomic(lk.owner == -1){         lk.owner = -1;
     lk.owner = pid;            }
  }
}
```

Figure 7. Locks implemented with conditional atomics.

4.2 Concurrency Primitives

The key novelty in PSKETCH is the support for synthesizing concurrent programs. To write these concurrent sketches, we included three concurrency constructs in the PSKETCH language. While these three constructs are standard, supporting them required rethinking our synthesis algorithm, which we discuss in Section 6.

Threads. Threads are created with the construct **fork** (**int** i, N) b which spawns N threads and blocks until all N threads terminate. Each thread executes the statement b. The index variable i contains a unique id for each thread, from 0 to $N - 1$. All variables declared inside b are thread-local. All other variables, together with the heap, are shared.

Our current system only supports programs with a single **fork** statement, optionally preceded by a sequential prologue and followed by a sequential epilogue. However, this limitation is not inherent to the method; it is only a matter of engineering to extend the system to support multiple, nested **fork** statements.

Atomic Sections. An atomic section is a block of code that is guaranteed to execute without interference from other threads. Atomic sections can be used to model the atomic primitives, such as compare-and-swap or read-and-increment, available on a particular architecture.

Synchronization. PSKETCH translates all synchronization primitives into conditional atomic sections [13]. A conditional atomic is an atomic section that blocks until its condition holds. For example, lock and unlock primitives can be implemented in terms of conditional atomics as shown in Figure 7.

It is worth noting that PSKETCH does not support spin-locks, so they must be modeled with conditional atomics. We discuss this limitation in more general terms in Section 6.

4.3 Specifications in PSKETCH

In SKETCH, a sketch is synthesized into a program that complies with a separately provided behavioral specification, which is bound to the sketch with the **implements** keyword. The synthesizer either outputs a program is functionally equivalent to the specification (in terms of observable outputs) or reports that the sketch cannot be resolved (*i.e.*, cannot be completed to behave like the specification).

[1] PSKETCH does not necessarily resolve **reorder** so that it minimizes mutual exclussion. If optimality is desired, we believe the best way to achieve it is to synthesize many correct candidates and select the best one by measuring the performance of each, as is done in autotuning [6]. Still, the programmer can use assert statements to constrain solutions to only those with mutexes that are, *e.g.*, separated by at most two statements.

This mode of specification is still supported in PSKETCH, but it is useful only for those parallel sketches for which one expects deterministic behavior. The final state of concurrent data structures typically depends on the nondeterministic interleaving of operations in concurrent threads, so a specification defined by input/output equivalence is less useful.

PSKETCH allows the programmer to specify desired correctness conditions using **`assert`** statements. The semantics of PSKETCH is that the synthesized program must (1) behave like the specification bound with the **`implements`** clause; and (2) be free of assertion failures on all inputs and all thread interleavings. These assertions also include implicit ones added by the synthesizer to guarantee memory safety and freedom from deadlock. The programmer-specified correctness criteria are typically checked in the epilogue; Section 8 describes how we used assertions to define correctness for some of the benchmarks we evaluated.

5. Synthesis for Sequential Sketches

The SKETCH synthesizer solved sequential sketches using a counterexample-guided synthesis algorithm. The algorithm was introduced in [20], where it was presented as a solver for the problem of 2-quantifier Quantified Boolean Satisfiability specialized for synthesis of sketches. We have recently found a deep connection between the original algorithm and inductive program synthesis [4]. This section describes the original algorithm from this more general perspective as counterexample-guided inductive synthesis (CEGIS), and highlights the connection to program verification. The algorithm is described on a reduced subset of the language that is limited to basic control flow and integer holes. Section 6 extends this algorithm to handle concurrent sketches that use conditional atomic sections as the only synchronization primitive. Section 7 describes how the remaining language features are implemented in terms of these basic constructs.

5.1 The Counterexample-Driven Inductive Synthesis

A sketch with only integer holes can be understood as a parametrized program $Sk[\mathbf{c}]$, where $\mathbf{c}$ is a *control* vector containing the values for all the integer holes in the program. For a given input x, we can represent the correctness requirements for candidate $Sk[\mathbf{c}]$ as a predicate $P(x, \mathbf{c})$ on x and $\mathbf{c}$. Thus, the sequential sketch synthesis problem reduces to finding a control vector satisfying the following equation.

$$\exists \mathbf{c}.\forall x.P(x, \mathbf{c}) \tag{5.1}$$

The two-quantifier alternation makes this problem very difficult, but the CEGIS algorithm solves it by using the principle of inductive synthesis.

The problem of inductive synthesis is to generate a candidate implementation that is consistent with a set of observations about the behavior of the program on a given set of inputs. For sequential sketch synthesis, our observations consist of a set of inputs E, together with the observation that the candidate $Sk[\mathbf{c}]$ must satisfy the correctness criteria on these inputs. Therefore, we can frame the inductive synthesis problem as follows.

$$\exists \mathbf{c}.\forall x \in E.P(x, \mathbf{c}) \tag{5.2}$$

Given a boolean representation of the predicate P, Equation (5.2) can be expanded and supplied to a SAT solver directly since the universal quantification over the small set E can be expressed as a simple conjunction.

On its own, however, an inductive synthesizer is unable to guarantee the correctness of the candidate solution. The synthesizer can only guarantee that the resulting implementation will match the given observations. As more observations are added, the resulting implementations are expected to converge to a correct implementation, but the inductive synthesizer is unable to detect convergence on its own.

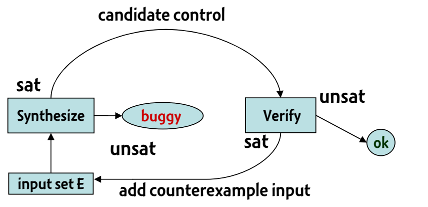

Figure 8. Counterexample driven synthesis algo.

To address this problem, CEGIS couples the inductive synthesizer with a verification procedure. The verifier serves two functions: it rejects incorrect candidates until convergence is reached, and it produces observations to drive the inductive synthesizer. The verifier is very good at producing observations because every time a candidate fails, the counterexample that proves the failure is guaranteed to cover a corner case not covered by any previous observations. The complete algorithm is illustrated in Figure 8. It is worth noting that the CEGIS algorithm places very few requirements on the verification procedure; any verification procedure capable of producing concrete counterexamples can be incorporated into the algorithm.

The power of the CEGIS algorithm was demonstrated empirically in [20]; for example, in one reported experiment, the sketch solver synthesized a sketch of AES by analyzing only 655 inputs from space of 2^{256} possible inputs. This demonstrated both the power of the inductive synthesis approach, and the high quality of the observations produced by the verifier.

The challenge of concurrent synthesis is to extend this algorithm for the case when the observations are no longer just inputs, but traces showing how specific thread interleavings in a candidate solution lead to property violations.

6. Synthesis for Concurrent Sketches

This section develops a concurrency-aware synthesizer to support the concurrency extensions to the SKETCH language. The concurrent synthesizer exploits the benefits of inductive synthesis observed in the sequential setting. In that setting, synthesis from observations allowed us to ignore all but a few counterexample inputs, which turned the 2QBF synthesis problem into a sequence of SAT problems. Here, we show that a correct candidate can be computed by considering only a few counterexample thread interleavings, sidestepping the need to reason about all possible thread interleavings during synthesis. We implemented the algorithm on top of the existing SKETCH infrastructure, using SPIN as our verification engine with very positive results.

The algorithm follows counterexample-guided inductive synthesis:

- The inductive synthesizer evaluates each candidate on a set of observations. Each observation is a fixed thread schedule. As a result, the inductive synthesizer evaluates each candidate only on traces induced by the observations, ignoring all other interleavings. Traces have sequential semantics, so observations reduce the concurrent synthesis problem into a sequential one.
- The verifier is standard in that it considers all thread interleavings in the provided candidate. If the candidate is bad, the verifier generates a counterexample trace that witnesses the assertion violation. Counterexample traces are then used as observations.

The algorithm can accommodate any verifier as long as it produces a bounded counterexample. The correctness guarantees of the system will be those which the verifier can decide. However, the inductive synthesizer can only eliminate candidates based on violation of safety properties on a trace. Therefore, we require that any liveness properties be approximated as safety properties which must hold after a bounded number of steps. For example, the synthesizer enforces termination by requiring that candidates terminate after a bounded number of steps for the bounded inputs it considers.

The algorithm outlined above is relatively straightforward. The challenge is how to turn a trace into a valid observation; that is, how to make it applicable to all candidates. Compared to inputs, which act as observations in the sequential setting, traces do not lend themselves directly to that role: while a counterexample input produced on one candidate is applicable to other candidates, a trace is specific to a candidate and thus incompatible with others. We could, of course, arbitrarily project a trace onto another candidate, but we want *good* observations. An observation is good if it prunes away many bad candidates, by exposing their violations. Since a trace exposes an (unidentified) problem detected in a candidate by a verifier, it is desirable that projected trace retains the ability to expose this problem in candidates that share the problem.

Two issues complicate projection of traces onto other candidates.

1. *Projection onto candidate space.* We cannot afford to project traces individually for each candidate, as there are too many candidates. Instead, we need to transform the sketch so that a given counterexample trace is projected simultaneously onto all candidates in the candidate space.
2. *Preservation of errors.* We know that a counterexample trace exposes an error in a candidate, but we do not know what aspects of the trace caused the error. Hence, projection cannot aim to preserve a specific fragment of a trace. Instead, it needs to preserve as much as possible under some notion of "error in the candidate."

Let us now address the first problem, starting from first principles. To turn a trace into an observation, we need to project a trace t_{c_1} produced on a candidate program c_1 onto a trace t_{c_2} valid on a candidate c_2. We denote the projection of t_{c_1} onto c_2 with $t_{c_1} \triangleright c_2$, or $t \triangleright c$ when the origin of t is clear from the context. We define $t_{c_1} \triangleright c_2$ to be a single trace, rather than a set of traces. (The latter would enable preservation of more errors present in t_{c_1}, but this would come at the cost of growing the size of the observation set.)

The goal is to be able to use a set T_e of counterexample traces to find a plausible candidate c. A candidate is *plausible* iff trace $t \triangleright c$ does not fail, for all traces $t \in T$. A trace is considered to *fail* when it encounters an assertion violation or a deadlock. This is denoted with a predicate $\mathit{fail}(t)$. Thus, we want to search for a candidate c to satisfy the following equation.

$$\forall t \in T_e \,.\, \neg \mathit{fail}(t \triangleright c)$$

In order to make the search efficient, we need to produce a boolean encoding of the problem above, similar to what is used in sequential synthesis [20]. The first step, is to encode the space of candidate programs as a function $Sk[\mathbf{c}]$ parametrized by a bit-vector $\mathbf{c}$, so different values of $\mathbf{c}$ make Sk a different candidate.

The second step is to create a new function $Sk_t[\mathbf{c}]$ such that

$$Sk_t[\mathbf{c}] = t \triangleright Sk[\mathbf{c}]$$

$Sk_t[\mathbf{c}]$ is therefore a projection of trace t on the candidate $Sk[\mathbf{c}]$. With this encoding, the SAT solver is now left to solve the problem

$$\forall t \in T_e \,.\, \neg \mathit{fail}(Sk_t[\mathbf{c}])$$

Where $\mathit{fail}(Sk_t[\mathbf{c}])$ is a boolean function of $\mathbf{c}$ computed symbolically for each individual trace in T_e. Note that $Sk_t[\mathbf{c}]$ is a sequential trace, so this is the same inductive synthesis problem as in the sequential setting.

To explain how $Sk_t[\mathbf{c}]$ is computed symbolically from Sk and t, we need to return to the second question—how to preserve errors exposed by a trace. To simplify the presentation, let us assume that the sketch is acyclic, which implies that all candidates are acyclic. This is not a serious simplification because our inductive synthesizer explores executions of bounded length. The acyclic restriction is will simplify the explanation because each statement is executed at most once.

When projecting traces, our goal is to ensure that, whenever possible, the projected trace preserves the error exposed in the original trace. An error is that aspect of a candidate program that was responsible for an assertion violation: it could be not enough synchronization causing a race condition, or too much of it causing a deadlock, or any other "bug" allowed by the sketch. An error is an inherently vague concept, so rather than defining it directly, we define preservation of errors in terms of maximally preserving the ordering of steps in the original trace. A step is a pair (s, i), of a statement s and the thread i which executed it.

We say that a trace t' *preserves* a trace t if all steps common to t' and t are executed in the same order in both traces; *i.e.* if s_1 precedes s_2 in t, and both s_1 and s_2 are present in t', then s_1 precedes s_2 in t'.

This notion of preservation is practically relevant if preserving step ordering indeed preserves the conditions that lead to an error. This is to be expected if preserving the order also helps preserve the dataflow relationship that led to the error in the first trace. For example, if data flowing from s_1 caused an assertion failure in s_2 in a candidate c_1, then if a trace for candidate c_2 preserves this error-causing dataflow, the trace should serve to eliminate candidate c_2. Preserving the order does not necessarily mean the dataflow will be preserved. For example, it is possible that executing s_1 before s_2 exposed an error in c_1 but it does not in c_2, because something in the program c_2 executed between these two steps and masked the error. However, step-ordering preservation is simple to enforce and we hypothesize that it preserves many errors. Preserving ordering of statements worked well for the errors that we manually examined.

Therefore, in general, we want $Sk_t[\mathbf{c}]$ to be a *preserving* projection of t. However, it is not always possible to find a preserving projection of a trace into another candidate program, as the following example illustrates.

```
bool c = ??;
thrd1: { sa; if (c) wait; s1; if (!c) signal }
thrd2: { sb; if (!c) wait; s2; if (c) signal }
```

This sketch corresponds to two candidate programs, selected based on the value of c:

c_t: `thrd1: { sa; wait; s1; } thrd2: { sb; s2; signal; }`
c_f: `thrd1: { sa; s1; signal; } thrd2: { sb; wait; s2; }`

All traces for c_t execute s_2 before s_1. However, none of these traces can be projected onto c_f in a preserving manner because all traces for c_f execute s_1 before s_2.

In these situations where a preserving projection is not possible, we require that $Sk_t[\mathbf{c}]$ be a preserving projection of the longest prefix of t for which such a projection is possible. For example, if the trace t_t for c_t is $(2, sb)$,$(1, sa)$,$(2, s2)$,$(2, signal)$,$(1, wait)$,$(1, s1)$, then the encoder can not produce a complete preserving projection into c_f. Thus, $Sk_{t_t}[0] = (2, sb), (1, sa)$, a projection of the longest prefix of t_t for which a preserving projection is possible.

The algorithm that produces $Sk_t[\mathbf{c}]$ from Sk and t is relatively simple. As a first step, it performs if-conversion [1] on the sketch

Sk to turn it into a sequence of predicated atomic statements (either atomic blocks, or simple assignments). An interesting property of this representation is that any candidate implementation $Sk[\mathbf{c}]$ derived from the sketch will contain a subset of the statements present in Sk.

In the second step, the algorithm produces a version of the sketch Sk^i for each thread i in the trace. The n^{th} statement of Sk^i, s^i_n is derived from the n^{th} statement of Sk by renaming all local variables to have thread-unique names. This guarantees that local variables will behave as expected when we interleave statements from different threads.

The next step is to interleave the sequences of statements corresponding to the different threads into a single sequence of statements. To do this, the algorithm sorts all the statements s^i_n according to the partial order imposed by both the thread and the sequential threading; namely: (i) If step (s_i, n) precedes (s_j, m) in the trace, then $s^i_n < s^j_m$. (ii) If $i = j \wedge n < m$ then $s^i_n < s^j_m$. (iii) If the trace t exposes a deadlock involving a set of steps $D = (s_j, m) \ldots$, then if s^j_m corresponds to a step in the deadlock set, and s^i_n does not, then $s^i_n < s^j_m$.

The last constraint is there for technical reasons, to make it easier for us to both do deadlock detection and rule out suffixes of traces which can not be made into a preserving projection. The final step in producing Sk_t is to take the sequence of guarded statements form the previous step, and replace all conditional atomics **atomic**(c) s with a conditional like the one shown below.

```
if(c)
   s;
else
   if(some other thread can make progress)
      return OK;
   else
      assert 0 : ‘‘deadlock’’;
```

The resulting encoding Sk_t represents the preserving projection of the trace t onto all the candidates in the space. Moreover, $fail(Sk_t[\mathbf{c}])$ can be represented as a boolean function of **c**, which allows us to solve the inductive synthesis problem efficiently with a SAT solver. Section 8 will describe our empirical evaluation of the method, but before that, we must describe how the synthesizer handles the high-level sketching constructs.

7. Translating Sketching Constructs

In the last two sections, we described how the synthesizer completes sketches containing only integer holes. In this section, we describe how the new high-level sketching constructs are implemented by showing their translation to simple code fragments with integer holes.

7.1 Regular Expression Generators

The translation of RE-generators depends on whether the RE-generator is an l-value or r-value. In both cases, we will use the following terminology. Assume that the RE-generator r describes a set $S(r)$ of k syntactically valid strings, denoted $s_1, \ldots, s_k$.

Translating an r-value RE-generator is straightforward. This translation requires $\lg k$ bits of primitive holes.

```
translateRvalueGen ($r$) =
   switch (??) {
      case 1: return $s_1$;
      ...
      case $k$: return $s_k$;
   }
```

The translation of an l-value RE-generator r that appears in the statement r=e is much like the r-value RE-generator translation, except that each statement in the **switch** block is a choice of assignments from e to s_i.

7.2 Reorder Statement

A **reorder** block with a set S of k statements $s_0, \ldots, s_{k-1}$ represents $k!$ possible candidate programs; so the synthesizer needs to encode this exponential space of possibilities in a reasonable amount of PSKETCH code. The PSKETCH synthesizer actually contains two different encodings for the reorder block, each with different tradeoffs of space and complexity.

Our first, quadratic, encoding is shown below. It uses $k \lg k$ control bits and, after unrolling the **for** loop, will have k copies of each statement in the block, for a total of k^2.

```
translateReorder ($S$) =
   int[$k$] order = ??$^k$
   assert noDuplicates in order
   for (i=0 to $k-1$)
      switch (order[i]) {
         case 1: $S_1$
         ...
         case $k$: $S_k$
      }
```

Notice that the assert forces the synthesizer to consider only semantically legal values of the array order (permutations of $1 \ldots k$), which is initialized with k primitive holes.

The second encoding actually requires exponential space, but for many sketches, it has proven to be significantly more efficient than the quadratic one. The basic idea is as follows. Suppose that we start with a list of m statements $s_0; \ldots; s_{m-1}$, and we want to insert a statement s_m somewhere in the list. We can encode this easily in 2*m+1 statements.

```
i=??;
if(i=0){ $s_m$;} $s_0$;
if(i=1) {$s_m$;} $s_1$;
...
if(i=m-1){ $s_m$;} $s_{m-1}$;
if(i=m) $s_m$;
```

We can apply this construction recursively to build a representation of the **reorder**. To do this, we start with s_0, and we use the construction above to add s_1 before or after it. Then, we repeat the process to insert s_2 into the resulting sequence; the same process is repeated to insert each subsequent statement. The resulting representation will have 2^i copies of s_i, and will require on the order of n^2 control bits.

Surprisingly, for many benchmarks this encoding is much better than the quadratic one, both in terms of speed *and* size. There are several reasons for this. First, in most of our benchmarks, the number of statements in the reorder blocks are relatively small. Moreover, our reorder blocks often contain statements of drastically different sizes; blocks with only a couple of very expensive statements can be encoded more efficiently with the exponential encoding. For example, if a reorder block has two expensive statements and three inexpensive ones, the quadratic encoding will require 10 expensive statements and 15 cheap ones. With the exponential encoding, we can encode this block with 3 expensive statements and 28 cheap ones, as long as we add them in the right order. Thus, if the expensive statements are more than twice as expensive as the cheap ones, the exponential encoding will be more efficient.

8. Evaluation

This section presents our evaluation of the desugaring of the PSKETCH language shown above, the new PSKETCH language introduced in Section 4, and our CEGIS algorithm from Section 6.

Specifically, we evaluate the performance of our PSKETCH compiler and the expressiveness of the PSKETCH language on a suite of benchmarks. The benchmarks were chosen because they are complex to implement, due to subtle issues caused by concurrency. Our performance results are encouraging:

- PSKETCH successfully searched spaces of about 10^8 syntactically unique candidates in under an hour, consuming less than 500 MiB of memory.
- Our CEGIS algorithm required only a few observations (meaning only a few calls to the verifier) to resolve a sketch, or determine that it could not be resolved. In our benchmarks, PSKETCH required 10 iterations to find a correct implementation from a space of about 10^8 possibilities. PSKETCH was also able to show after only 7 observations that one of our benchmark sketches could not be resolved.

The expressiveness of the PSKETCH language is harder to evaluate, but we show example sketches of our benchmarks below and argue that they capture the insight behind a solution, with a minimum of unnecessary detail. For example, we were able to sketch and synthesize a previously-unknown-to-us `Dequeue()` method of a lock-free queue. These results indicate that parallel programmers might find PSKETCH useful.

We begin this section by introducing our hypotheses about the PSKETCH system. Next, we present the benchmarks we used to teste these hypotheses, showing some of our example sketches. We then report PSKETCH's performance across our test suite, and discuss the results. Finally, we summarize the limitations we encountered in the PSKETCH synthesizer.

8.1 Hypotheses

We wish to evaluate the following hypotheses.

Synthesis scales well with the size of the candidate program space. This scalability is the key to the sketching approach: it enables programmers to write sketches with less mundane or subtle detail, leaving its completion to the synthesizer. We test this hypothesis by measuring the time for PSKETCH to resolve sketches that encode increasingly large candidate spaces.

Our encoding of the observations made from failed candidates captures useful information about the cause of failure. This appraises the projection strategy we use to encode information from the traces in the inductive synthesizer. The number of observations required to resolve a sketch (or show that it cannot be resolved) can measure the strategy 's effectiveness. Fewer observations suggest that the encoding is capturing more useful information from each trace.

The PSKETCH *language is expressive for this domain*. We do not attempt to measure this quantitatively; instead, we show how we expressed the insights behind our benchmarks using PSKETCH.

8.2 Benchmarks

Our benchmarks are intended to represent various sketching scenarios across different problems. These sketches were chosen as exemplars; we have sketched other data structures that we omit here, including a doubly-linked list and full version of the lazy list-based set described below. Table 8.2 summarizes the more detailed descriptions of the benchmarks that follow.

8.2.1 Lock-free queue

The first version of this queue, `queueE1`, contains a sketch of a restricted version of the `Enqueue()` method discussed in Section 1. It is restricted in that its candidate space is smaller than the `Enqueue()` sketch from Section 1. The second version, `queueE2`, has the full `Enqueue()` sketch shown in Section 1.

Sketch	Description	$\|C\|$
`queueE1`	Lock-free queue: restricted `Enqueue()`	4
`queueE2`	Lock-free queue, full `Enqueue()`	10^6
`queueDE1`	`queueE1`, plus sketched `Dequeue()`	10^3
`queueDE2`	`queueE2`, plus sketched `Dequeue()`	10^8
`barrier1`	Sense-reversing barrier, restricted	10^4
`barrier2`	Sense-reversing barrier, full	10^7
`fineset1`	Fine-locked list, restricted `find()` method	10^4
`fineset2`	Fine-locked list, full `find()`	10^7
`lazyset`	Lazy list, singly-locked `remove()`	10^3
`dinphilo`	Approximation of dining philosophers problem	10^6

Table 1. Summary of benchmark sketches. C is the set of candidate programs encoded by each sketch.

For this benchmark, we also analyzed the complexity of resolving a problem where multiple methods had been sketched. The `Dequeue()` sketch from Section 1 had too few holes to serve this purpose. Instead, one of us decided to try implementing `Dequeue()` with a single while loop. In a few minutes, he wrote the very simple sketch shown below. The sketch simply places in a reorder block all the statements that one could reasonably expect to be necessary for the solution. The solution times for this experiments correspond to the `queueDE1` and `queueDE2` benchmarks.

```
Object Dequeue() {
  QueueEntry tmp = null;
  boolean taken = 1;
  while (taken) {
    reorder {
      tmp = {| prevHead(.next)?(.next)? |};
      if (tmp == null)
        return null;
      prevHead = {| (tmp|prevHead)(.next)? |};
      if (!tmp.taken)
        taken = AtomicSwap(tmp.taken, 1);
    }
  }
  return tmp.stored;
}
```

The queue benchmarks were resolved with respect to the conjunction of the following correctness conditions:

- *Sequential consistency* [15]. If a thread A enqueues a_1 and a_2, then a_1 must be dequeued before a_2. Note that is a weaker condition than linearizability [12].
- *Structural integrity*. The queue is not corrupted by concurrent operations. Specifically: (1) the head and tail are not **null**; (2) `prevHead.taken == 1`; (3) the tail is reachable from the head; (3) `tail.next == null`; (4) there are no cycles in the queue; (5) no "untaken" nodes precede "taken" ones.

PSKETCH also enforces memory safety by default: no **null** pointers may be deferenced, and array accesses must be within bounds. It is worth noting that for `queueE2` and `queueDE2`, we found that we had to use more than one operation per thread or more than two threads for verification in order to get solutions that generalized to more threads and more operations per thread.

8.2.2 Sense-reversing barrier

Barriers allow multiple threads to synchronize at the same program point before continuing. They are difficult to implement for a couple of reasons. First, the last thread to reach the barrier must realize that it is last, then awaken the other, waiting threads. Second, barriers must prevent newly awoken threads from passing through later barrier points.

An insight to solving these problem is to separate consecutive barrier points into two phases, even and odd; the phase is called the barrier's "sense," and reverses after each barrier point [10]. The barrier object keeps the global boolean sense, and each thread has a local sense. When a thread reaches a barrier point, it either waits until its local sense matches the barrier sense, or if the last thread, reverses the barrier sense, awakening the waiting threads.

However, this insight is far from an implementation. The barrier code requires subtle reasoning about interleaved threads and intermediate barrier states. We claim that the PSKETCH language is well suited to capturing the insight behind a sense-reversing barrier. Below, we sketch the barrier's `next()` method. The sketch encodes `next()` as a "soup" of operations, to be executed (or not) under some conditions on the barrier state. The synthesizer is left to find an implementation that avoids harmful races, deadlocks, and other intricate details.

We first write the fields of the `Barrier`: (1) `sense`, the current phase; (2) `senses`, the local senses of each thread; and (3) `count`, the number of threads yet to reach the barrier. We define the soup of operations comprising the insight behind `next()` as follows:

1. Update the thread's own sense of the barrier.
2. Atomically decrement the count of threads yet to arrive.
3. Under some condition, wait until the barrier sense changes to some predicate of the thread's own sense.
4. Under some condition, set the barrier's sense and yet-to-arrive count so as to wake up the other threads, and prepare the barrier for the next shot.

Before finishing the sketch, we define "under some condition" as a PSKETCH generator function that returns a boolean expression of its arguments:

```
boolean predicate (a, b, c, d) {
  return {| (!)? (a==b | (a|b)==?? | c | d) |};
}
```

Now, translating the operations above into a sketch is straightforward. We make them into a "soup" by placing them in a **reorder** block:

```
void next (Barrier b, Thread th) {
  boolean s = b.senses[th];
  s = predicate (0, 0, s, s);
  int cv = 0;
  boolean tmp = 0;
  reorder {
    // (1) Update t's local sense
    b.senses[th] = s;
    // (2) Decr. count of yet-to-arrive threads
    cv = AtomicReadAndDecr (b.count);
    // (3) Wake up other threads, reset barrier
    tmp = predicate (b.count, cv, s, tmp);
    if (tmp) {
      reorder {
        b.count = N;
        b.sense = predicate (b.count, cv, s, s);
      }
    }
    // (4) Wait at barrier
    tmp = predicate (b.count, cv, s, tmp);
    if (tmp) {
      boolean t = predicate (0, 0, s, s);
      atomic (b.sense == t);
    }
  }
}
```

The benchmark `barrier2` is the sketch shown above. The companion `barrier1` is a reduced version with a smaller candidate program space. The barrier's correctness was established by a client program that ensured that threads always joined properly at each barrier point, together with the implicit deadlock check performed by PSKETCH. This client program launched N threads that reached a barrier B times. Before waiting at the b^{th} invocation of `next()`, each thread t set a bit `reached[t][b]`. After passing through the b^{th} call to `next()`, each thread ensured that its left neighbor t_l had also reached the b^{th} barrier by asserting `reached[t-l][b]`.

8.2.3 Finely locked, list-based set

This data structure implements the `Set` data type with a sorted, singly linked list. In a highly concurrent setting, locking the entire list for each `add()`, `remove()`, and `contains()` operation is unacceptable. The insight behind the "finely locked" list is to maintain a sliding window of locks around the nodes being traversed during set operations, to allow concurrent modifications to disjoint areas of the list. Implementing this locking scheme, known as hand-over-hand-locking [11], is difficult; the programmer must order the acquisition and release of locks while traversing the data structure, keeping in mind deadlocks and data structure corruptions due to concurrent modifications.

For this list, we sketched a method `find (key)` that returns `cur`, the node with a least key greater than or equal to `key`, and `prev`, the node greatest key less than `key`. The main loop of the `find` method was described in Section 4; the sketch left the synthesizer to decide which nodes to lock and unlock and under what conditions, and how to order these locking, unlocking, and traversal statements. It is straightforward to implement the other data structure methods using this `find()` helper. The benchmark `fineset2` is our full sketch, and `fineset1` is a reduced version of `fineset2`. The correctness criteria for these benchmarks were similar to those for the `queue*` suite, with structural checks specific to this structure.

8.2.4 Singly-locked remove() method of lazy list

This is a problem proposed by [11]. Its basis is a lazily-updated, list-based set data structure due to [9]. The `add()` and `remove()` methods of this set are optimistic, in that they traverse the data structure without locking. Only when the list is to be modified do they check that the their view of the list is still valid. Both `add()` and `remove()` acquire two locks before modifying the list.

This problem asks whether the list's `remove()` method can be modified to take only one lock, instead of two (the answer is "no"). We translated this problem into a sketch for PSKETCH to solve by first removing the **lock** statements from the original `remove()` method. Next, we gave PSKETCH the freedom to **lock** any one of a set of nodes at any point in the body of the stripped-down `remove()`, and likewise for **unlock**. The correctness criteria for this sketch were the same as for the `fineset*` benchmarks.

When we ran this benchmark with two threads performing both `add` and `remove`, the synthesizer returned "NO", as expected. Surprisingly, PSKETCH *was* actually able to find a solution that worked for the case where one thread performs only adds and another thread performs only removes.

8.2.5 Dining philosophers

This problem has P philosophers at a circular table, with a plate of spaghetti in the center. A philosopher needs two chopsticks to eat. Each philosopher has chopsticks at his left and right, but because the table is circular, there are only P total chopsticks. The problem is to find a chopstick-acquisition policy which avoids deadlock, in which no philosopher can eat; and starvation, in which particular philosophers cannot eat. Thus, we want a resource policy that satisfies the properties (1) some philosopher can always eat; and (2) every philosopher will always eventually eat.

We modeled the problem in PSKETCH as follows: there are P philosophers encoded as a `fork(int p; P)` block, each contending for its left and right of P locks. The philosophers attempt to eat T times, blocking if they cannot acquire their left and right chopsticks. The resource acquisition policy was sketched as an expression of t, p, P, which indicated whether the right or left chopstick should be acquired first. The order in which the chopsticks were released was also left unspecified. As to correctness, PSKETCH implicitly enforces property (1) above by ensuring that the execution is deadlock free. As we described earlier, we can only enforce livenes properties by approximating them as a safety property in a bounded execution. Our sketch approximates property (2) by ensuring that all philosophers are able to eat T times in the $P * T$ steps of the execution. With this sketch and this correctness conditions, the synthesizer was able to produce a correct implementation of the protocol; a minor variant over the standard solution presented in textbooks [16].

8.3 Performance

We tried to synthesize each benchmark for workloads with various numbers of threads and operations, and patterns of operations when possible. The particular tests of the `queue*`, `fineset*`, and `lazyset` benchmarks are labeled with the following scheme: a test named $ed(ed|ed)$ means that first a sequential enqueue e was performed , next a sequential dequeue d, and finally two threads were forked to each perform an enqueue then dequeue $(ed|ed)$. The set tests use the same scheme, with a and r standing for “add” and “remove”, respectively. For each test, we gathered the following data:

- *Resolvable* – whether the sketch could be completed into a correct implementation.
- *Itns* – the number of observations required for CEGIS to terminate.
- S_{solve}, V_{solve}– for the synthesizer, the time for its SAT solver to return SAT or UNSAT; for the verifier, the time for SPIN to complete its search for a counterexample schedule.
- S_{model}, S_{model}– for the synthesizer, the time to build a boolean satisfiability problem; for the verifier, the time to compile an input model into a verification program.
- *Time:Total* – total elapsed time between invoking PSKETCH and it returning an answer. This time does not equal S_{solve}+ S_{model}+ V_{solve}+ S_{model} because part of the time is spent in our compiler frontend.
- *Memory* – the maximum memory used by the synthesizer, verifier, and PSKETCH. The maximum total memory includes memory used by our Java frontend.

We tested PSKETCH on a laptop with a 2 GHz Core 2 Duo processor and 2 GB of RAM, running version 2.6.20-16 of the Linux kernel. The results are tabulated in Figure 9.

The data in Figure 9 reveal a few interesting trends. First, there is an approximately linear correlation between the $\log$ of the size of the candidate space C and number of iterations before finding a solution, as observed in a sequential sketch synthesizer [20]. We have plotted $\log C$ against number of iterations in Figure 10, for selected tests. Second, neither synthesis nor verification clearly dominated the total solution time across the test suite, though verification tended to be more expensive. Third, we see that for each benchmark, changing the number of threads or the methods called on each thread had a big effect on verification times, but synthesis stayed fairly constant. A fourth trend is the large amount of time needed to generate and compile the SPIN verifiers, which dominated the total time for several tests.

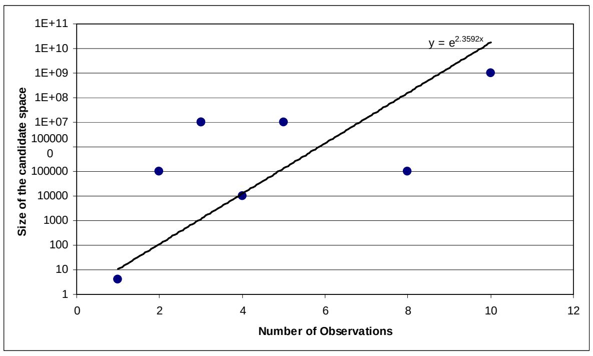

Figure 10. Candidates Vs. Observations

8.3.1 Limitations

PSKETCH’s most severe limitation is that it only guarantees correctness of synthesized programs with respect to safety properties, up to a bounded number of executed instructions. However, we believe that with future work, PSKETCH can handle liveness properties. A second limitation is that PSKETCH only returns a single correct implementation of a sketch. In many contexts, one wishes to find *all* correct solutions, then search these for an optimal one (*e.g.*, with autotuning [6]). The CEGIS algorithm can trivially produce multiple correct candidates, but future research might additionally guide its search by optimality criteria.

PSKETCH is also hampered by engineering limitations, mostly due to the delicate connection between the SPIN verifier and the SAT synthesizer. As mentioned above, for some programs we saw a large discrepancy between the time needed to verify the unsimplified models emitted by PSKETCH and hand-simplified versions of the same models. Applying traditional compiler optimizations to these models was difficult, because they threatened to upset the correlation of counterexamples between SPIN and our synthesizer. Another practical problem was the large overhead of compiling SPIN verifiers, which were C programs with up to tens of thousands of lines of code. Both problems are amenable to better engineering.

Synthesizing data structures that are correct with respect to linearizability is a future goal. Our current CEGIS algorithm can synthesize and verify data structures with respect to linearizability criteria, but it is difficult to embed these criteria in sketches. We believe that this problem can be solved with richer specifications in the PSKETCH language.

9. Related Work

In previous work we have proposed sketching as a methodology for developing efficient algorithms from a low-level outline thereof. This line of work proved useful for writing bit-streaming programs [19] and was later extended to work for arbitrary finite computations [20] and unbounded stencil computations [18]. The sketching approach is related to earlier research in control inference, including foundational work on Prolog [14], as well as efforts in the field of AI for determinizing an agent’s behavior via learning techniques [3]. Alternatively, transformational synthesis frameworks (*e.g.*, [8, 17]) are largely domain-specific and apply separately provided programmer insights through an interactive synthesis process.

Synthesis of Concurrent Algorithms. While computer-aided verification of concurrent programs has gained significant momen-

	Test	Resolvable	Itns	Time (s)					Maximum Memory (MiB)		
				Total	S_{solve}	S_{model}	V_{solve}	V_{model}	Total	S_{mem}	V_{mem}
`queueE1`	$ed(ee\|dd)$	yes	1	8.79	0.01	0.04	0.07	5.55	54.41	13.72	5.13
	$ed(ed\|ed)$	yes	1	9.24	0.02	0.04	0.86	6.1	67.04	13.73	8.25
	$(e\|e\|e)ddd$	yes	1	13	0.05	0.12	5.67	5.05	72.81	17.54	31.69
`queueDE1`	$ed(ee\|dd)$	yes	4	46.97	2.63	4.76	0.32	31.95	135.51	54.7	6.69
	$ed(ed\|ed)$	yes	4	64.18	5.27	7.98	7.09	33.76	172.92	66.73	22.31
`queueE2`	$ed(ed\|ed)$	yes	5	114.7	16.22	9.93	5.32	71.98	171.69	69.31	17.63
	$(e\|e\|e)ddd$	yes	8	249.2	44.74	23.3	104.59	60.98	213.69	92.64	114.69
`queueDE2`	$ed(ed\|ed)$	yes	10	3091.37	2676.07	147.07	16.28	184.72	489.26	313.2	30.13
`barrier1`	$N = 3, B = 2$	yes	4	49.74	0.11	0.57	37.3	8.07	177.31	17.54	130.31
	$N = 3, B = 3$	yes	8	120.21	0.39	2.37	97.03	14.69	398.19	19.85	331.06
`barrier2`	$N = 2, B = 3$	yes	9	66.46	4.375	13.613	1.272	35.243	153.67	54.73	10.70
`fineset1`	$ar(ar\|ar)$	yes	2	130.44	2.5	4.21	2.55	110.97	161.14	55.46	23.88
	$ar(ar\|ar\|ar)$	yes	1	363.89	0.56	1.03	279.02	74.29	249	29.03	169.38
	$ar(a\|r\|a\|r)$	yes	1	196.52	0.73	1.25	112.02	73.86	153.56	29.17	73.88
	$ar(arar\|arar)$	yes	1	165.43	0.66	1.26	80.02	73.85	259.25	29.14	136.38
	$ar(aaaa\|rrrr)$	yes	2	225.54	8.63	12.94	74.12	111.07	345.62	156.81	145.75
`fineset2`	$ar(ar\|ar)$	yes	3	281.46	13.41	15.17	4.03	229.24	260.14	123.77	34.81
	$ar(ar\|ar\|ar)$	yes	3	795.19	12.95	20.58	509.59	232.38	376.63	149.32	233.44
	$ar(a\|r\|a\|r)$	yes	2	384.83	11.57	13.7	170.42	171.1	325.26	169.07	95.75
	$ar(arar\|arar)$	yes	2	299.97	4.85	6.33	99.82	174.01	346.56	75.68	212.94
	$ar(aaaa\|rrrr)$	yes	3	468.7	40.86	46.3	107.69	228.61	563.1	287.41	227
`lazyset`	$ar(aa\|rr)$	yes	12	179.17	5.32	16.6	11.43	107.4	294.03	54.28	11.38
	$ar(ar\|ar)$	NO	7	100.24	1.88	5.41	2.51	66.49	246.81	41.87	9.81
`dinphilo`	$N = 3, T = 5$	yes	4	34.03	4.34	4.39	6.22	12.61	194.08	114.33	19.19
	$N = 4, T = 3$	yes	3	54.46	1.96	2.23	36.11	9.93	158.69	53.15	78.75
	$N = 5, T = 3$	yes	3	745.94	3.06	2.99	724.5	10.2	1419.5	83.98	1340.31

Figure 9. Performance results.

tum in recent years, the automated *synthesis* of concurrent algorithms is a relatively new research direction, and most of the previous work in the field is designed for synthesis within in a specific domain of algorithms (*e.g.*, [5]). Notable in this context is the recent work on synthesis of concurrent garbage collectors by Vechev *et al.*. In an earlier work [23] the authors apply an automated transformational-style space exploration to derive provably correct variants from a basic (correct) concurrent GC implementation. In a more recent work [24] an exhaustive exploration procedure is applied to a space of implementationstion variants with varying degrees of atomicity and instruction reordering, and combined with effective pruning of vacuously incorrect implementation sub-spaces. In this approach the authors deploy a separate verification procedure based on the SPIN model checker [13] to check the absence of concurrency bugs in each of the generated candidate implmenetations. Their framework, unlike ours, is capable of verifying concurrent implementations that manipulate arbitrary unbounded data structures, thanks to the use of abstraction in the verification procedure. This, however, is not an inherent limitation of our approach and the use of abstraction-capable verifiers is a work in progress. Also, the generation method used in their approach heavily depends on tailored semantic rules to prune the search space effectively, and is restricted to a predefined set of concurrency-related transformations and synchronization primitives. In contrast, our synthesizer applies generic transformations to reduce the problem into its 2QBF representation and delegates the effort of conducting an effective search to an efficient, general purpose SAT-based solver.

Verification of Concurrent Data Structures. Particular concurrent data structures are often checked for correctness using automated provers. Examples include the verification of a prominent wait-free concurrent set implementation [9, 22]. Such efforts often rely on massive proof scripts and associated domain-specific logic (*e.g.*, in PVS or some other proof system) that need to be written per verification task. In contrast, our framework can be used to synthesis *and* automatically verify arbitrary concurrent implementations with only few assumptions about the underlying execution model. CheckFence [7] is a tool that can find subtle concurrency bugs occurring under various memory consistency models and generates a counter example that can be used to infer the apporpriate fix (*i.e.*, adding memory fences to enforce consistency). Similar to our approach, checking an imperative concurrent program is reduced to a SAT problem, and as such bears similar limitations. It is different from ours in the way that imperative programs are encoded into Boolean circuits. In recent work [2], verifying linearizability of concurrent heap-manipulating algorithms was done using 3-valued logic abstraction. Here, an abstract interpreter (TVLA) was applied to capture the (finite) differences between states exhibited by two implementations of the same data structure, and to verify their unification at linearization points. Although sound and highly expressive, this framework requires apriori knowledge of the linearization points in a concurrent implementation, and is known to have inherent scalability problems due to the size of the abstract domain that is being used.

10. Conclusion

The paper describes a new sketch synthesizer for the development of concurrent programs, with an emphasis on concurrent data structures. Sketching affords the programmer the same fine control over the structure of the resulting program as manual coding, while at the same time allowing him to leave unspecified those parts of the program which are hard to derive by hand.

Our system relies on a CEGIS algorithm to generate candidate implementations by analyzing traces of failed implementations to try to prevent the new candidates from exhibiting the same bugs. To our knowledge, ours is the first synthesizer capable of using counterexample traces from failed concurrent programs to guide the search for a correct implementation.

We implemented PSKETCH relying on the SPIN verifier and the SKETCH synthesis infrastructure. With the system, we have sketched and synthesized concurrent data structures including a Lock-free queue, and a list with hand-over-hand locking. In each of these programs, the tricky fragments were exclusively sketched and successfully synthesized.

Acknowledgment

We want to thank Gilad Arnold for his help with preparation of the submission manuscript. This paper was influenced by the many discussions we had with Martin Vechev, Eran Yahav, and Mooly Sagiv. We are grateful to the anonymous referees for their helpful comments. This work is supported in part by the National Science Foundation with grants CCF-0085949, CNS-0326577, and CNS-0524815, a generous gift from IBM Corporation, an IBM Fellowship, and the University of California MICRO program.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, New York, NY, USA, 1983. ACM.

[2] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV '07: 19th International Conference on Computer Aided Verification*, volume 4590, pages 477–490. Springer, 2007.

[3] D. Andre and S. Russell. Programmable reinforcement learning agents. *Advances in Neural Information Processing Systems*, 13, 2001. MIT Press.

[4] D. Angluin and C. H. Smith. Inductive inference: Theory and methods. *ACM Comput. Surv.*, 15(3):237–269, 1983.

[5] Y. Bar-David and G. Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 305–305, New York, NY, USA, 2003. ACM.

[6] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology. In *International Conference on Supercomputing*, pages 340–347, 1997.

[7] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, volume 42, pages 12–21, New York, NY, USA, 2007. ACM.

[8] B. Fischer and J. Schumann. Autobayes: a system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 13(3):483–508, May 2003.

[9] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS '05: 9th International Conference on Principles of Distributed Systems*, volume 3974, pages 3–16. Springer, 2005.

[10] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[11] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[12] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[13] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[14] R. Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, 1979.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[16] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[17] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[18] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, volume 42, pages 167–178, New York, NY, USA, 2007. ACM.

[19] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 281–294, New York, NY, USA, 2005. ACM Press.

[20] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *ASPLOS '06*, San Jose, CA, USA, 2006. ACM Press.

[21] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*, pages 404–415, New York, NY, USA, 2006. ACM Press.

[22] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 129–136, New York, NY, USA, 2006. ACM.

[23] M. T. Vechev, E. Yahav, and D. F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 341–353, New York, NY, USA, 2006. ACM.

[24] M. T. Vechev, E. Yahav, D. F. Bacon, and N. Rinetzky. Cgcexplorer: a semi-automated search procedure for provably correct concurrent collectors. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 456–467, New York, NY, USA, 2007. ACM.