Quantitative Synthesis for Concurrent Programs

Pavol Černý, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna and Rohit Singh

IST Austria (Institute of Science and Technology Austria)

Am Campus 1

A-3400 Klosterneuburg

Technical Report No. IST-2010-0004

http://pub.ist.ac.at/Pubs/TechRpts/2010/IST-2010-0004.pdf

October 7, 2010

Quantitative Synthesis for Concurrent Programs

Pavol Černý IST Austria
Krishnendu Chatterjee IST Austria
Thomas A. Henzinger IST Austria
Arjun Radhakrishna IST Austria
Rohit Singh IIT Bombay

Abstract

We present an algorithmic method for the synthesis of concurrent programs that are optimal with respect to quantitative performance measures. The input consists of a sequential sketch, that is, a program that does not contain synchronization constructs, and of a parametric performance model that assigns costs to actions such as locking, context switching, and idling. The quantitative synthesis problem is to automatically introduce synchronization constructs into the sequential sketch so that both correctness is guaranteed and worst-case (or average-case) performance is optimized. Correctness is formalized as race freedom or linearizability.

We show that for worst-case performance, the problem can be modeled as a 2-player graph game with quantitative (limit-average) objectives, and for average-case performance, as a $2\frac{1}{2}$-player graph game (with probabilistic transitions). In both cases, the optimal correct program is derived from an optimal strategy in the corresponding quantitative game. We prove that the respective game problems are computationally expensive (NP-complete), and present several techniques that overcome the theoretical difficulty in cases of concurrent programs of practical interest.

We have implemented a prototype tool and used it for the automatic synthesis of programs that access a concurrent list. For certain parameter values, our method automatically synthesizes various classical synchronization schemes for implementing a concurrent list, such as fine-grained locking or a lazy algorithm. For other parameter values, a new, hybrid synchronization style is synthesized, which uses both the lazy approach and coarse-grained locks (instead of standard fine-grained locks). The trade-off occurs because while fine-grained locking tends to decrease the cost that is due to waiting for locks, it increases cache size requirements.

1. Introduction

Developing concurrent programs that harness the power of modern multi-core machines is a difficult and error-prone task, as witnessed by a number of errors found in published algorithms [6, 21], and in production code (see e.g. [7]). We focus on *partial program synthesis*, where the programmer specifies the sequential parts of the program, while leaving out the synchronization mechanisms. The synthesis algorithm automatically chooses the right synchronization mechanisms to ensure correctness of the resulting program. We apply this approach to concurrent data structure implementations. Here, on one hand it is easier to program the sequential accesses to the data structure in imperative style than to provide a declarative specification. On the other hand, it is easier to specify the concurrency aspects of correctness (for example, requiring race freedom or linearizability) than to insert synchronization constructs manually. Thus the partial program synthesis makes best of both worlds available to the programmer.

Example. We illustrate why performance measures are necessary in synthesis. Consider two threads accessing a buffer with 2 cells. The buffer consists of two cells, `x` and `y`. The value 0 indicates that a cell is empty. The threads can call the `store` method which checks if one of the cell of the buffer is empty, and if so, stores the input value in an empty cell and returns `true`. If both of the cells are full, `store` returns `false`. Conversely, a the method `load`, returns a value of a nonempty cell, if one of the cells is nonempty and returns 0 otherwise.

A sequential sketch for `store` is in Figure 1. The programmer specifies that synchronization would be performed using locks, but

```
choice C1 : {global.lock(); xlock.lock(); ylock.lock();
global.unlock(); xlock.unlock(); ylock.unlock();skip;}
  public boolean store(byte input)
1:    choice C1; //should be global.lock() or xlock.lock();
2:    if (x == 0)
3:        x = input;
4:        choice C1; //should be global.unlock() or xlock.unlock()
5:        return true
6:    choice C1; //should be skip or xlock.unlock()
7:    choice C1; //should be skip or ylock.lock()
8:    if (y == 0)
9:        y = input;
10:       choice C1; //should be global.unlock() or ylock.unlock()
11:       return true
12:   choice C1; //should be global.unlock() or  ylock.unlock()
13:   return false
```

Figure 1. Producer-Consumer Sketch

does not specify how and when to lock/unlock, and whether to use a global lock, or fine-grained (cell-local) locks. The sequential sketch for `load` is similar to the sketch of `store` method. We require data-race-freedom, i.e., there should not be two potentially simultaneous accesses to the same location, with one of them being a write access.

Examining the sketch in Figure 1, we see that the sketch allows three types of implementations. We describe a representative of each category.

- incorrect implementations: implementation S_1 that does not use any locks. This leads to data races on both `x` and `y`. (Other incorrect implementations might leave some lock locked.)
- global locking: implementation S_2 obtained by choosing `global.lock()` at line 1, `global.unlock()` at lines 4, 10 and 12, and `skip` at other choice locations.
- cell-local locking: implementation S_3 obtained by choosing `xlock.lock()` at line 1, choosing `xlock.unlock()` at lines 4 and 6, choosing `ylock.lock()` at line 7, choosing `ylock.unlock()` at lines 10 and 12.

However, the two correct implementations are not equivalent with respect to performance in all cases. We try to distinguish and choose between them using a *performance model*. In order to define a performance model, we first specify what (concurrency-related) events will be taken into account: for this example we choose lock access (locking or unlocking) and *idling* (the oppurtunity-cost of not using an available processor). We consider two performance models: model P_1 which assigns a cost c to idling, and a cost 0 to lock access, and a model P_2 which assigns a cost 0 to idling and a cost c to lock access.

If the performance model is P_1, implementation S_3 which allows more concurrency by using fine-grained locks will perform better than S_2 which uses a global lock. However, under performance model P_2, the situation would be reversed — S_2 uses fewer locks and therefore performs better. Therefore for performance model P_1, the synthesizer should return S_3 as its result, and for performance model P_2, the synthesizer should return S_2 as its result.

In general, performance measures need to include a (rudimentary) model of the architecture of the target machine, and different implementations might be more or less suitable for a particular ar-

chitecture. For example, if we assume a two-core processor, the performance damage from using global locks might not be high enough to justify using fine-grained locks, which has a higher cache requirement and lowers performance gains obtained from per core cache. Hence a synthesis framework should support quantitative, performance-related objectives for synthesis.

Methods. In this paper, for the first time, we synthesize concurrent implementations of data structures with respect to quantitative performance models. The input consists of (1) a sequential sketch, (2) a performance model, (3) a usage model, (4) a scheduling model, and (5) a bound on the size of the client programs. The *sequential sketch* is a sequential implementation of a data structure in the form of a set of methods that access shared variables or shared heap. The sequential sketch leaves out synchronization code but includes "holes", which are to be filled in by the synthesizer using a specified set of synchronization constructs. We say that a program is *allowed* by a sketch if it can be obtained by filling holes of the sketch by synchronization constructs. We distinguish between *finite sketches* (sketches with finite memory), and *heap-accessing sketches* (sketches that access a shared singly-linked heap with potentially unbounded size). The heap-accessing sketches we consider are required to satisfy the constraints of the method automaton model introduced in [8]. This enables us to develop algorithmic analysis for correctness and performance.

The second input to the synthesis problem is a *performance model*, given by a weighted automaton. The automaton has edges for actions, and assigns different costs to actions such as locking, idling and context-switching. The automaton model allows to assign costs based on past sequence of actions, for example, if a context-switch happens soon after the preceding one, then its cost might be lower. Thus our model allows to specify complex cost models, e.g. cache access. The third input, *usage model* specifies how frequently each method of the sequential sketch will be used. For example, for the case of a set based data structure with *contains, add* and *remove* methods, it is known [17] that in practice, the *contains* method is called 90% of the time, *add* 9% and *remove* 1%. Providing this information to the synthesizer enables it to prefer implementations that give better performance to the contains method. The fourth input is a *scheduling model* that every time schedules one of the the active threads. Our schedulers are state-based models, and hence support flexible scheduling schemes (e.g., a thread waiting for long may be scheduled with higher probability). In performance analysis, the average-case analysis is as natural as worst-case analysis. For the average-case (randomized) analysis, a *probabilistic scheduler* is needed.

The fifth input, the bound on the size of clients, specifies which clients (programs that access the data structure) the synthesizer considers. We consider two cases: (1) For finite sketches, our approach limits the clients to have a statically bounded number of threads but we allow clients where each thread can call an unbounded number of methods. The bound on the size of clients is therefore given by (n), the number of threads and the synthesis the considers the most general clients on n threads. The most general client on n threads is one in which every thread executes any number of the methods in any order. (2) For heap-accessing sketches, we allow clients with only a bounded number (n) of threads, and furthermore, each thread can call only a bounded number (m) of methods. More general clients for heap-accessing sketches are not considered as checking linearizability for such clients is undecidable [8]. Correctness is specified using one of two generic conditions in our approach: data-race-freedom and linearizability. Data-race freedom for finite state sketches and can be specified using an assertion on the global state space. For heap-accessing sketches P, checking linearizability can be reduced to checking reachability on a finite-state system via a lockstep construction [8]. The correctness

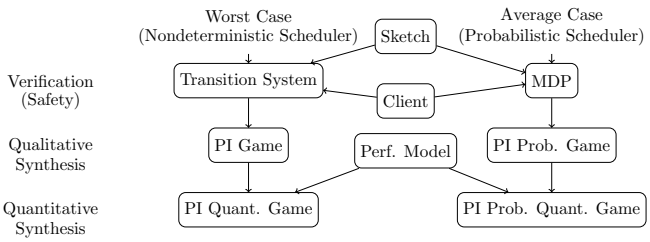

Figure 2. Synthesis Flow, where PI stands for Partial Information

check is performed on clients with a bounded number of threads each of which calls a bounded number of methods. However, we prove a *cut-off* theorem for a class of programs (that includes our main case study of concurrent list implementations). It shows that in order to check linearizability for programs with an unbounded number of threads, it is sufficient to check clients with 2 threads each of which calls 1 method.

The output of synthesis is an implementation P of the concurrent data structure, such that (a) it is allowed by the sketch, (b) it is correct with respect to clients conforming to the bound, and (c) it has the best performance of all programs satisfying (a) and (b) with respect to the performance, usage and scheduling models.

The flow for quantitative synthesis is illustrated in Fig 2. Given a sketch and a client, we obtain a transition system model. In presence of probabilistic schedulers we require the more general model of Markov decision processes that have probabilistic transitions. The probabilistic scheduler is necessary to model the average case performance. We then show that the problem of synthesis of sketches is naturally modeled as a two-player partial-information game: one player is the *synchronizer* who picks the synchronization constructs offered in the sketch and the opponent player chooses the worst possible input and schedules. In presence of probabilistic schedulers we require partial-information games with probabilistic transitions. We then show that the performance-aware synthesis problem can be solved through optimal strategies of partial-information games with quantitative objectives (limit-average or mean-payoff objectives). If the game graphs are unbounded (as is the case for heap-accessing model), then we show how to extend the correctness of lock-step construction from transition systems with qualitative objectives [8] to games with quantitative objectives. The quantitative synthesis problem gives rise to a new game theoretic problem and we show that the problem is NP-complete.

We present several techniques that overcome the theoretical difficulty of NP-hardness, and works well for the class of examples we study. Our first step is an optimized version of the lock-step reduction that significantly reduces the state space of the game graphs, and this key step makes the synthesis feasible in practice. Our second steps are efficient strategy elimination techniques: (a) our first elimination is based on a light-weight partial correctness check for strategies, by which we are able to reduce the set of strategies significantly; (b) the second elimination is based on counter-example analysis: if we obtain a counter-example to witness that a strategy is not a correct solution, we use the counter-example to rule out further strategies to be explored. With the above two methods we prune the strategies by three-orders of magnitude.Our third step is novel algorithmic and practical optimization techniques for quantitative evaluation of correct strategies. For quantitative analysis we require solution of MDPs with quantitative objectives. Our example MDPs were very large, but sparse, and we came up with optimization techniques for the special class of our MDPs that lead to an order of magnitude of improvement in the running time.

Results. In order to evaluate our synthesis algorithm, we have implemented a prototype tool and applied it to a finite sketch example of producers and consumers accessing a buffer, and to two heap-

accessing sketches of concurrent lists. In all of the examples, the sketch considered can give rise to incorrect programs, correct-but-inefficient programs, and correct-and-efficient programs. The performance parameters that we used for these examples were the cost of locking, cost of idling, and cost of a context-switch. For list sketches, our method automatically synthesizes classical synchronization schemes for implementing a concurrent list, such as fine-grained locking or a lazy algorithm for certain relative costs of parameters. For other parameter values, a new, hybrid synchronization style is synthesized, which uses both the lazy approach and coarse-grained locks (instead of standard fine-grained locks).

Summary. We summarize our main contributions: (1) (a) we show how to use game theoretic framework to model the synthesis of concurrent programs (to the best of our knowledge, this is the first application of partial-information games for synthesis of concurrent data structures); (b) we explicitly use generic performance, usage, and scheduling models, and thus provide a very flexible framework for synthesis with quantitative measures of performance; and (c) we show how games with quantitative objectives provide solution to the synthesis problem of our framework. (2) We present a cut-off theorem for linearizability of concurrent lists with locks that enables reducing the problem of checking linearizability for programs with unbounded number of threads to checking linearizability programs with bounded number of threads, where each thread calls only a bounded number of methods. (3) We present optimized abstraction techniques for state space reduction for our game graphs, and present several novel algorithmic and practical optimization techniques for our MDPs with quantitative (limit-average) objectives. (4) We have implemented a prototype and applied it to case studies of synthesizing implementations of concurrent lists. Further technical proofs and details omitted due to lack of space can be found in [1].

Related works. The problem of synthesis from specification was originally posed by Church [10]. Synthesis for synchronization constructs is also an old problem and the celebrated paper [11] presented an algorithm for synthesis of synchronization skeletons. Synthesis of reactive systems was considered in [22]. In contrast to our work, all these works focused on qualitative synthesis without any performance measure. Recent works have considered quantitative synthesis [5, 9], however the focus of these works has been the synthesis of sequential systems from temporal logic specifications.

Sketching for bounded domains programs and concurrent data structures has been studied in [25] and [24] respectively. However, none of the above works consider performance-aware algorithms. Abstract interpretation based synthesis was presented in [27], and is only optimal with respect to the number of interleavings.

Verifying correctness of concurrent data structures has received a lot of attention recently. For example, [20], unlike this work and [8], considered only a bounded heap. Static analysis methods based on shape analysis [3, 23] are not completely automated. A recent approach by [26] automatically is able to linearizability for unbounded heaps and threads but, reports limited success with concurrent lists.

The inclusion of the usage model in our synthesis algorithm was inspired by works which introduced concurrent set algorithms [16, 17], where the performance of the algorithms is analyzed for various usage models.

2. Method Sketches

We define a model for programs accessing shared memory, either bounded or unbounded (consisting of a bounded number of boolean variables or consisting of an unbounded heap). Method sketches are an extension that adds nondeterministic sketch states to the method automaton model introduced in [8]. We recall here basic definitions and notations. We refer the reader to [8] for more details.

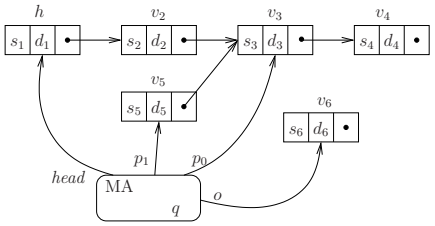

Figure 3. Singly-linked data heap and a method automaton

Singly-linked data heaps. Let D be an unbounded set of data values equipped with equality and linear order $(D, =, <)$ and let Σ be a finite set of symbols. A *singly-linked data heap* L is a tuple $(V, next, flag, data, h)$, where V is a finite set of vertices, $next$ is a partial function from V to V, *flag* is a function from V to Σ, $data$ is a function from V to D, and $h \in V$ is the initial (head) vertex. The heap L can be naturally viewed as a labeled graph with edge relation $next$. L is *well-formed* if this graph has no cycles reachable from h. Figure 3 shows an example heap with six vertices.

Method sketches. A *method sketch* M is a tuple $(Q, Sk, B, PV, DV, T, q_0, F, head, O_C)$, where Q is a finite set of locations, Sk is a subset of Q, B is a finite set of shared boolean variables, PV is a finite partially-ordered set of pointer variables, DV is a finite set of data variables, T is a set of transitions (explained below), $q_0 \in Q$ is the initial location, $F \subseteq Q$ is a set of final locations, $head$ is a pointer constant, and O_C is a set of pointer constants. The set of transitions T is a set of tuples of the form (q, g, a, q'), where $q, q' \in Q$ are locations, g is a guard, and a is an action. There are no outgoing transitions from the final locations.

A method sketch represents a sketch of a program that operates on a heap. The set Sk contains locations where the program is only sketched, that is, these locations contain a number of choice transitions, several of which can be enabled at a given time. The locations in Sk are called *sketch locations*. A method sketch is called *finite* if its set of pointers PV is empty (it thus does not access the heap). A method sketch is called *heap accessing* if it is not finite (i.e., $PV \neq \emptyset$).

A *method automaton* is a method sketch with an empty set Sk. A method automaton M' is *allowed* by a method sketch M if M' can be obtained from M by omitting all but one of the outgoing transitions for all locations from Sk, and by setting the set Sk to empty set in M'. A *sketch* is a set of method sketches. A *program* is a set of method automata. A program P is *allowed* by a sketch $\mathcal{P}$ if each method automaton in P is allowed by a corresponding method sketch in $\mathcal{P}$.

A method sketch operates on a singly-linked data heap L. The pointer variables range over $V \cup \{nil\}$, where nil is a special value, and are denoted by p, p_0, p_1, etc. Let $\leq_{PV}$ be the partial order on PV. The partial order is required to have a minimum element, denoted by p_0. The variable p_0 is called the *current pointer*, and the other variables in PV are called *lagging pointers*. The constant $head$ points to the vertex h and is shared across method sketches. The pointer constants in the set O_C (denoted by e.g. o, o_0, o_1) give method sketches input/output capabilities and are referred to as *IO pointers*. The set R of pointers (i.e. pointer variables and pointer constants) of a method sketch is defined by $R = PV \cup \{head\} \cup O_C$. The data variables in DV range over the domain D.

The *guard* g include symbol and pointer equality comparison and data equality and order comparisons. Let succ_{PV} successor relation defined by the partial order $\leq_{PV}$. The *actions* consists of updates to pointers, variables, and flags and the updates must

satisfy the following restrictions: for updates (i) $next(p) := p'$ and (ii) $p := p'$, we must have $\text{succ}_{PV}(p', p)$. The restriction $\text{succ}_{PV}(p', p)$ enforce that the heap is traversed in a monotonic manner. This necessitates that pointer variables are statically ordered, and the furthest pointer can be assigned to the next of its vertex, but lagging pointers can be assigned only to a pointer further up in this ordering. Fields of vertices, including the next field, corresponding to lagging pointers can be updated. Also, the three fields of vertices (flag value, data value, and the next pointer) can be updated together atomically (this is needed for encoding some of the Java concurrency primitives). For further details see [8].

We require the actions of a method sketch to satisfy the "*One write before move*" (OW) restriction. This restriction states that there is at most one action modifying $flag(p)$, at most one action modifying $data(p)$, and at most one action modifying $next(p)$ performed between two successive changes of the value of the pointer variable p. The restriction can be enforced syntactically — we omit the details. We note that the restriction OW holds for every implementation we have encountered.

A method sketch is *deterministic* iff given a location that is not a sketch location and a valuation of variables, at most one transition is enabled. A method automaton is *deterministic* iff given a location and a valuation of variables, at most one guarded action is enabled. Notice that a method automaton allowed by a deterministic method sketch is a deterministic method automaton.

Figure 3 shows a method automaton in location q. Its $head$ pointer points to the vertex h of the heap. A client of the automaton can store values in the vertex v_6 pointed to by the IO pointer o. The variables p_0 and p_1 are pointer variables of the method automaton.

Examples. We illustrate the model by showing how it captures synchronization primitives and other core features of concurrent data structure algorithms.

- *Inserting a vertex.* Assume that the position to insert the vertex has been found - the new vertex pointed by o is to be inserted between p_1 and p_0. The transition relation can then include $(q, true, next(o) := p_0, q_1)$ and $(q_1, true, next(p_1) := o, q_2)$.
- *Locking individual vertices.* We can model locking of vertices using the Σ value. Let us suppose that $\Sigma = \{u, l_1, l_2, \ldots\}$, for unlocked, locked by thread 1, locked by thread 2, etc. Locking is captured by the transition: $(q_0, flag(p) = u, flag(p) := l_1, q_1)$ for thread number 1. Unlocking can be modeled as follows: $(q_1, flag(p) = l_1, flag(p) := u, q_2)$.

Client programs. A *client program* composes a finite set of method sketches (or method automata) sequentially and in parallel. They are defined by the following grammar rules: $E ::= E_S \mid (E_S \parallel E)$ and $E_S ::= M \mid (M\ ;\ E_S)$, where M is a method sketch or a method automaton. This corresponds to a number of threads composed in parallel, with each thread containing a sequential composition of methods.

Inputs. Given a method sketch M, a triple (L, sB, io), where L is a singly-linked data heap, sB is a valuation of the shared boolean variables of B, and io is a valuation of the IO pointers of M is called an *input* to a method. A method input is *well-formed* if L (interpreted as a graph) is acyclic. In the remainder of the paper, we will assume that method inputs are well-formed. (For the case of finite method sketches, there are no special boolean inputs - these can be directly part of the shared heap.)

Given a client program, its *set of inputs* is composed of the (shared) singly-linked data heap L, the valuation of sB boolean variables, and for each method sketch (or automaton) that is a part of the client, a valuation of its IO pointers.

Correctness conditions for client programs. The correctness for client programs that access shared memory is specified using one of two generic conditions — data-race-freedom and linearizability. Data race freedom requires that there should not be two potentially simultaneous access to the same memory location, with at least one of the two being a write access. We use this condition for finite method sketches. It can be specified as an assertion on the global state space. Linearizability [18] is the standard correctness condition for concurrent data structure implementations. We use linearizability as a correctness condition for heap-accessing sketches. The specific instantiation of linearizability that applies to method automata is described in [8].

Performance automaton. We define a flexible and expressive performance model via a weighted automaton that specifies costs of actions. A *performance automaton* W is a tuple $W = (Q, \Sigma, \delta, q_0, \gamma)$, where Q is a set of states, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, q_0 is an initial location and γ is a cost function $\gamma : Q \times \Sigma \times Q \to \mathbb{Q}$. The labels in Σ represent (concurrency-related) actions that incur costs, while the values of the function γ specify these costs. The symbols in Σ are matched with the actions (edge symbols) performed by the system to which the performance measures are applied. There is a special symbol in Σ, denoted by ot, that signifies that no action of the ones we are tracking occurred. The costs that can be specified in this way include for example: (i) the cost of locking (ii) the cost of context switches (iii) idling, that is the penalty paid if a physical core is idling, due to threads being blocked by locks.

An example specification that uses the three costs mentioned above is the automaton W in Figure 4. The automaton describes the costs for specific type of concurrency-related operations (locking (l), context-switching (cs), and the cost of limiting concurrency. For the latter, the automaton specifies an *idling* cost w, i.e. cost of a thread being prevented from running (by a synchronization mechanism), while there is still an unused physical core. The example assumes there are 4 physical cores, so a thread can block at most three threads that otherwise could run.

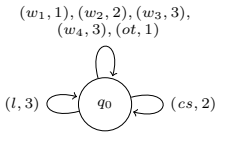

Figure 4. Perf. Aut.

Specifying the costs via a weighted automaton is more general than only specifying a list of costs. Let us assume that we model the effects of cache size in the cost of a context switch. (Note that this is possible, as the major part of a context switch costs comes from cache-related preemption delays.) Specifying the cost model as an automaton allows for example specifying that the cost of a context-switch is lower if the number of steps since the context switch is low, as the performance damage from cache-related preemption delays is low in this case. There are other possible events that the performance model can keep track of, such as use of concurrency primitives other than locks. For the remainder of this paper we fix the alphabet $\Sigma = \{l, cs, w, ot\}$ that represent locking, context switches, and idling (waiting), however, our results hold for all deterministic performance automata.

Usage model. Given a sketch $\mathcal{P} = \{M_1, M_2, \ldots, M_k\}$, where $M_1, \ldots M_k$ are method sketches, a usage model is a function that for a method sketch M_i in $\mathcal{P}$ returns a rational number in $p_i \in [0, 1]$, such that $\Sigma_{1 \le i \le k} p_i = 1$. The usage model represents the relative frequency with which a particular method is called.

3. Games on Graphs for Quantitative Synthesis

In this section we first show how the verification problem of a program allowed by a sketch and a client can be modeled as transition systems, and as Markov decision processes in the presence of probabilistic schedulers. Then, we show how the synthesis problem can be modeled as a two-player partial-information game between the synchronizer and the adversary. We then show the correctness and performance analysis can be achieved through the solutions of

games with quantitative objectives, in particular, the optimal strategies of games correspond to a correct optimal program with respect to the performance and usage model. In general, the game graphs are infinite; we present abstractions to produce finite-state game graphs, and optimizations to reduce the game graph size to ensure synthesis is feasible. We finally study the complexity of these games, and show the problem is NP-complete.

3.1 Transition systems and MDPs from Programs

First, we present mathematical models of transition systems and Markov decision processes, and reduce the problem of verification of concurrent data structures to these models.

Definition 3.1. *(Transition systems and Markov decision processes). A* transition system $G = \langle S, A, \Delta, s_0 \rangle$ *consists of a set S of states, a finite set A of actions, an initial state s_0, and a* deterministic *transition function $\Delta : S \times A \to S$ that given a state s and an action a gives the successor state $\Delta(s, a)$. The graph (S, E) of the transition system consists of the set S of states, and the set $E = \{(s, t) \mid \exists a \in A.\ \Delta(s, a) = t\}$ of edges consists of the transitions. Let $\mathcal{D}(S)$ denote the set of probability distributions over S.* Markov decision processes (MDPs) *generalize transition systems with* probabilistic *transition function: an MDP $G = \langle S, A, \Delta, s_0 \rangle$ consists of the same components as a transition system, and $\Delta : S \times A \to \mathcal{D}(S)$ is a probabilistic transition function that given a state s and an action a gives the probability distribution $\Delta(s, a)$ over the successor states. The graph (S, E) associated with an MDP consists of the set S of states, and the set $E = \{(s, t) \mid \exists a \in A.\ \Delta(s, a)(t) > 0\}$ of edges consists of the positive probability transitions.*

Transition systems and MDPs from method automata and clients. Given a set of method automaton, a client and an initial state of the shared-memory, we define below the transition system of the program. The transition system is defined on the state space that is the set of all combined program and shared-memory states. In presence of probabilistic schedulers we will obtain MDPs.

Definition 3.2. *Given a program P, a client (method expression) $C = ((C_{1,1}\ ;\ \dots C_{1,k_1}) \parallel (C_{2,1}\ ;\ \dots C_{2,k_2}) \parallel \dots \parallel (C_{n,1}\ ;\ \dots C_{n,k_2}))$ for the program, and the set I of inputs for the methods, the* transition system $[\![P, C, I]\!]$ *is as follows:*

1. State space. *The state space of the transition system is the product of (i) the locations of the method automata $C_{i,j}$, (ii) the state space of local variable of $C_{i,j}$ and (iii) the set of heap configurations. The initial state is the state where the local variables of all $C_{i,j}$ are uninitialized, the locations of $C_{i,j}$ are the initial locations, and the heap is in the initial state specified by I.*
2. Actions. *The set A of actions of the transition system is the set of number of threads, i.e., $\{1, 2, \dots, n\}$.*
3. Transition function. *Given an action i, there is a transition edge from state s to t iff there is a method automaton $C_{i,j}$ and its guarded action (q, g, a, q') such that*
   (a) *for all $j' < j$, $C_{i,j'}$ is in the final location in s, i.e., all the automata in the sequential composition before $C_{i,j}$ have finished execution,*
   (b) *the guard g is true in s and the state of $C_{i,j}$ is q, and*
   (c) *the action a of method automata updating the state s produces t and the location of $C_{i,j}$ is q'.*

Schedulers. We now describe the role of schedulers in the execution of program-clients. Informally, a *scheduler* has a finite set of internal memory states Q. At each step, it considers all the active threads and chooses one either (i) non-deterministically (non-deterministic schedulers) or (ii) according to a probability distribution, which depends on the current internal memory state. We now describe how transition systems with probabilistic schedulers gives Markov decision processes. Given a transition system $[\![P, C, I]\!]$ and a scheduler Sch with states Q, we obtain the MDP $[\![P, C, I, \mathrm{Sch}]\!]^P$ as follows: The states of $[\![P, C, I, \mathrm{Sch}]\!]^P$ are the product of the states of $[\![P, C, I]\!]$ and Q. From a state (s, q) of $[\![P, C, I, \mathrm{Sch}]\!]^P$, the transition to (s', q') is possible (i.e., with positive probability) if there exists an i such that (a) when the program step corresponding to thread i operating on s produces s', and (b) scheduler Sch can schedule thread i and change its internal memory state to q'. The transition probability is the probability of Sch scheduling thread i in internal memory state q.

3.2 Partial-information Games from Sketches

Synthesis and games. The formal correctness analysis of a program P, a client expression, the inputs, and scheduler gives us the models of transition systems and Markov decision processes for verification. The more general problem of synthesis (automatically obtaining P from a sketch $\mathcal{P}$) is solved through a two-player game played on transition systems. Given a sketch $\mathcal{P}$ and a client (method expression) C, the two-player game is played on a transition system obtained in a similar fashion as $[\![P, C, I]\!]$, the only difference is that there are choices for synchronization that can be chosen, and then executed. The two-players in the game graph are the *synchronizer* (Player 1) who makes choices for the synchronization, and the *adversary* (Player 2) who makes choices for the inputs (and also for the scheduler if the scheduler is non-deterministic). In the game graph, there are several states that correspond to the same choice for Player 1. For example, two states may differ in the state of the local variables in a thread, but the thread location can be the same sketch location in both states. Hence, for a set of states Player 1 must make the same choice: this is modeled through an observation mapping, that maps states to observations, and given an observation, Player 1 must make the same choice. The choice for Player 2 at every state can be different. This gives rise to the notion of partial-information games played on game graphs where Player 1 has partial-information and Player 2 has perfect-information. We formally define them below.

Definition 3.3. *(Partial-information stochastic game graphs). A* partial-information stochastic *game graph is a tuple $G = \langle S, A, \Delta, (S_1, S_2), O, \eta, s_0 \rangle$, where (a) S is a finite set of states; (b) A is a finite set of actions; (c) $\Delta : S \times A \to \mathcal{D}(S)$ is the probabilistic transition function that maps every state $s \in S$ and action $a \in A$ to the probability distribution $\Delta(s, a)$ over successor states; (d) (S_1, S_2) is a partition of S into Player-1 and Player-2 states, respectively; (e) O is a finite set of observations; (f) $\eta : S \to O$ maps every state to an observation; and (g) s_0 the initial state. We will refer to these games as* PI $2\frac{1}{2}$*-player game graphs:* PI *for partial-information, 2 for the two players and $\frac{1}{2}$ for the probabilistic transitions. For a* PI $2\frac{1}{2}$*-player game graph G we associate the set E of edges as follows: $E = \{(s, t) \mid \exists a \in A.\ \Delta(s, a)(t) > 0\}$.*

Deterministic games. We now consider the following special cases of PI $2\frac{1}{2}$-player game graphs. If the transition function $\Delta : S \times A \to S$ is deterministic, rather than stochastic, then we have PI 2-player game graphs (partial-information deterministic games).

We now formally define how to obtain partial-information stochastic game graphs from a sketch $\mathcal{P}$, a client C and inputs I. We first present the formal definition of a probabilistic scheduler.

Definition 3.4. *Let $A = \{a_1, a_2 \dots a_n\}$ be a set of propositions. An n-thread* probabilistic scheduler *is an MDP with actions $\mathcal{A} = 2^A$ and a labelling $l : E \to \{1, 2, \dots, n\}$ of the edges of the MDP. We require that for a state s and an action a, the corresponding*

probability distribution $\Delta(s,a)$ has exactly one edge labelled i if $a_i \in a$ and no edge labelled i if $a_i \notin a$.

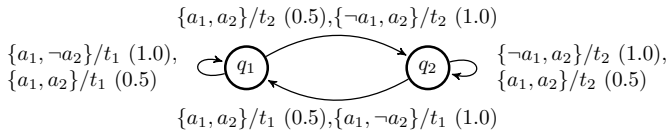

Figure 5. A uniform two-thread scheduler

Intuitively, each proposition a_i represents the fact that thread i is active and can proceed. An action represents the set of active threads and the label on an edge represents the thread to be scheduled next. An example of an 2-thread probabilistic scheduler is presented in Figure 5.

Definition 3.5. *Given a sketch $\mathcal{P}$, a client (method expression) $C = ((C_{1,1}\ ;\ \dots C_{1,k_1}) \parallel (C_{2,1}\ ;\ \dots C_{2,k_2}) \parallel \dots \parallel (C_{n,1}\ ;\ \dots C_{n,k_2}))$ for the sketch, a probabilistic scheduler* Sch *and the set I of inputs for the methods $C_{i,j}$, the* PI $2\frac{1}{2}$*-player game graph $\{\!\{\mathcal{P}, C, I, \mathrm{Sch}\}\!\}^P$ is defined as follows:*

1. *The* state space and initial state *are defined as before for transition graphs.*
2. State space partition. *The set of states of the game graph in which there exists method automaton $C_{i,j}$ which is at a state where a non-deterministic choice is to be made belong to S_1. The rest of the states are in S_2.*
3. Observations. *The set of Player 1 states in which the same non-deterministic choice is to be made are mapped to a single unique observation.*
4. Transition function.
   (a) *If s is a Player 2 state, then the transition function is similar to the one from transition graphs. It also includes transitions of the probabilistic scheduler. Let T_s be the set of threads active at state s and let m_s be the scheduler state in s. Given an thread $i \in T_s$, an edge from state s to t_i exists iff there exists a method automaton $C_{i,j}$ and its guarded action (q, g, a, q') such that:*
      i. *For all $j' < j$, the state of $C_{i,j'}$ is final in s,*
      ii. *The guard g is true in s and the state of $C_{i,j}$ is q, and*
      iii. *The action a of method automata updating the state s produces t and the state of $C_{i,j}$ is q'.*
      iv. *The state of the scheduler in t_i is m_{t_i} where m_{t_i} is the state of the scheduler from m_s on scheduling thread i on action T_s.*

   The probability of the edge (s, t_i) is the same as the probability of Sch *scheduling thread i in m_s on action T_s.*
   (b) *If s is a Player 1 state, there exists a choice of which sychronization action is to be performed by some thread (say i). Player 1 chooses the action and control moves to a state from which the only enabled action of thread i is the one chosen by Player 1. If there are more choices to be made, then this state is a Player 1 state; otherwise, it is a Player 2 state.*

A non-deterministic scheduler is similar to a probabilistic scheduler, the only difference is that for a state s and action a, the choice of the successors are non-deterministic, rather than probabilistic. In the above definition, if we consider non-deterministic schedulers, then the decision of which thread to execute is decided by the adversary (i.e., Player 2). Thus we get PI 2-player game graphs (rather than PI $2\frac{1}{2}$-player game graphs), and we denote them $\{\!\{\mathcal{P}, C, I\}\!\}$.

We show in the following discussion that the game-theoritic concepts relating to the transition game graphs correspond closely to the program execution, correctness and performance.

Plays. A PI $2\frac{1}{2}$-player game is played as follows: a token is placed on the initial state, and at every step, if the token is at a state in S_1, then Player 1 chooses an action, and otherwise, the token is at a state in S_2, and then Player 2 chooses an action. The token is moved to the successor state according to the action chosen by the players and the probabilistic transition function and the process of moving tokens generate a probability distribution over *plays*. An infinite sequence of steps (s_0, a_0, s_1), (s_1, a_1, s_2),... is represented as $\pi = s_0 a_0 s_1 a_1 s_2 a_2 \dots$ and is called a play. The set of all plays is denoted by Π.

Partial-information. The game is partial-information for Player 1, in the sense that she cannot observe the precise state where the token is in currently, but only the observation of the state. This is formalized as the notion of strategies.

Definition 3.6. *(Strategies and memoryless strategies). A* strategy *for Player 1 is a function $\tau_1 : O^* \to A$ that maps a sequence of observations to the next action. A play $\pi = s_0 a_0 s_1 a_1 s_2 a_2 \dots$ conforms to a Player 1 strategy τ_1 if for all $i \geq 0$, if $s_i \in S_1$, then $\tau_1(\eta(s_0)\eta(s_1)\dots\eta(s_i)) = a_i$. A* memoryless strategy *for Player 1 is independent of the history and depends on the current observation, i.e., a strategy τ_1 is memoryless if for all $w_1, w_2 \in O^*$ and $o \in O$ we have $\tau_1(w_1 o) = \tau_1(w_2 o)$, and hence τ_1 can be represented as a function $\tau_1 : O \to A$. Player 2 strategies and Player 2 memoryless strategies are defined analogously using states instead of observations (as Player 2 has perfect-information). The set of all Player i strategies is denoted as Γ_i. Given strategies $\tau_1 \in \Gamma_1$ and $\tau_2 \in \Gamma_2$ and the initial state s_0, (i) if the transition function is deterministic, then there is a unique play, denoted $\pi(\tau_1, \tau_2)$, that conforms both with τ_1 and τ_2; and (ii) if the transition function is probabilistic, then there is a unique probability measure, denoted $\Pr^{\tau_1,\tau_2}(\cdot)$, and $\mathbb{E}^{\tau_1,\tau_2}(\cdot)$ is the associated expectation measure.*

Fixing a memoryless strategy. Given a PI $2\frac{1}{2}$-player game graph $G = \langle S, A, \Delta, (S_1, S_2), O, \eta, s_0\rangle$, if a memoryless strategy τ_1 is fixed for Player 1, then we obtain an MDP, denoted $G_{\restriction \tau_1} = (S, A, \Delta_{\tau_1}, s_0)$, as follows: (a) for all $s \in S_2$ and $t \in S$, and for all $a \in A$ we have $\Delta_{\tau_1}(s,a)(t) = \Delta(s,a)(t)$; (b) for all $s \in S_1$ and $t \in S$, and for all $a \in A$ we have $\Delta_{\tau_1}(s,a)(t) = \Delta(s, \tau_1(\eta(s)))(t)$ (i.e., in Player-1 states the probabilistic transition for all actions is set according to the action chosen by τ_1). Similarly, if we fix a memoryless strategy in a PI 2-player game graph, then we obtain a transition system, and if we fix memoryless strategies for both Player 1 and Player 2 in PI $2\frac{1}{2}$-player game graphs, then we obtain a Markov chain.

In the transition graph $[\![P, C, I]\!]$, a play represents an execution of the client on inputs I, i.e., every step in a play corresponds to the execution of a particular instruction (guarded action) from a single method automaton in I. However, a similar statement that "every play in $\{\!\{\mathcal{P}, C, I\}\!\}$ represents the execution of the client C on I as a program P allowed by $\mathcal{P}$" does not hold. This is because at two different states corresponding to the same choice in the sketch, Player 1 may choose different statements to be executed next. However, if we restrict the strategies of Player 1 to memoryless-strategies, then the statement holds (however, there are still steps in plays which do not correspond to any instruction, but to choices made by Player 1). In fact, we show that there is a close correspondence between the memoryless strategies of Player 1 and the programs allowed by a sketch, i.e., we show in the following lemma that choosing a particular set of options in a sketch is equivalent to fixing a memoryless strategy for Player 1 in the transition game graph of the sketch. For the proof of the lemma we need the following definition of 0-closure.

Definition 3.7. *The* 0-closure *of a transition system (resp. an MDP) obtained by fixing a Player 1 memoryless strategy τ_1 in $\{\!\{\mathcal{P}, C, I\}\!\}$ (resp. $\{\!\{\mathcal{P}, C, I, \mathrm{Sch}\}\!\}^P$), respectively, is the transition system (resp. the MDP) obtained by removing the Player 1 edges that do not correspond to an actual execution step in the program, i.e., if $\{\!\{\mathcal{P}, C, I\}\!\}_{\restriction \tau_1}$ (resp. $\{\!\{\mathcal{P}, C, I, \mathrm{Sch}\}\!\}^P_{\restriction \tau_1}$) has an edge from s to t which does not correspond to any program instruction, in the 0-closure, edge (s,t) is removed and a new edges (s,u) are added for all $(t,u) \in E$. Also, an edge is added from all states where all methods in the client have finished execution to the start state. We denote the 0-closure as $\mathsf{ZC}(\{\!\{\mathcal{P}, C, I\}\!\}_{\restriction \tau_1})$ (resp. $\mathsf{ZC}(\{\!\{\mathcal{P}, C, I, \mathrm{Sch}\}\!\}^P_{\restriction \tau_1})$).*

Note that the 0-closure is well defined as we do not have Player 1 making infinite number of choices in a loop.

Lemma 3.8. *For all sketches $\mathcal{P}$, clients C, inputs I and schedulers* Sch*, the following assertions hold:*

- *For every program P allowed by the sketch $\mathcal{P}$, there exists a memoryless Player 1 strategy τ_1 such that $[\![P, C, I]\!] = \mathsf{ZC}(\{\!\{\mathcal{P}, C, I\}\!\}_{\restriction \tau_1})$ and $[\![P, C, I, \mathrm{Sch}]\!]^P = \mathsf{ZC}(\{\!\{\mathcal{P}, C, I, \mathrm{Sch}\}\!\}^P_{\restriction \tau_1})$.*
- *For every memoryless strategy τ_1 of Player 1 in $\{\!\{\mathcal{P}, C, I\}\!\}$, there exists a program P allowed by $\mathcal{P}$ such that $[\![P, C, I]\!] = \mathsf{ZC}(\{\!\{\mathcal{P}, C, I\}\!\}_{\restriction \tau_1})$ and $[\![P, C, I, \mathrm{Sch}]\!]^P = \mathsf{ZC}(\{\!\{\mathcal{P}, C, I, \mathrm{Sch}\}\!\}^P_{\restriction \tau_1})$.*

Inputs. To compute the correctness and performance cost of sketches, we parameterize the game graph based on the input to the methods and the scheduler. We define two simple modifications of game graphs to compute the worst-case and average-case performance of programs allowed by sketches. The correctness check and performance evaluation is over all possible inputs (and not just for a single input), and initial heap-states. This is achieved by giving the control of the inputs to the adversary of the synchronizer. This is achieved in the transition game graphs as described below. We present an informal description, which can be easily formalized.

1. The first transition where an input variable is read along a path from the start state is replaced by a sequence of transitions: First, Player 2 decides the value of the input variable and then, the actual transition which reads the input variable is executed.
2. A similar transformation is done to the transitions where a heap location is first read. Notice that each heap location is guessed separately and every guess may depend on the execution upto that point. This is captured by the strategies of Player 2 that are dependent on the history (not necessarily memoryless).

We denote the transition system and PI 2-player game graphs obtained this way as $[\![P, C]\!]$ and $\{\!\{\mathcal{P}, C\}\!\}$ respectively and their probabilistic versions as $[\![P, C, \mathrm{Sch}]\!]^P$ and $\{\!\{\mathcal{P}, C, \mathrm{Sch}\}\!\}^P$. A version of Lemma 3.8 easily follows for these versions of game graphs.

Note that giving the control of the inputs to the adversary has the following consequence: the transition system and game graphs becomes infinite for all heap-accessing programs, as an infinite number of heap configurations may be chosen as the initial heap configuration by the adversary. We will show that correctness and performance analysis of sketches are preserved under certain abstractions that produce finite transition systems and games.

3.3 Correctness and Performance analysis

Correctness Analysis. It follows from Lemma 3.8 that paths in the transition graphs of a sketch and a client correspond to a real schedule and execution of the corresponding client and that the transition game graph after fixing a memoryless strategy correspond to transition graphs for a single program allowed by the sketch. Therefore, we can check correctness conditions for programs allowed by sketches on transition game graphs. We model the correctness conditions as Safety objectives on the game graphs.

Safety objectives. A *safety* objective consists of a set B of bad states, and requires that states in B are never visited. In other words, the safety objective defines the subset Safety_B of the plays: for a play π, if never a state in B is visited, then $\pi \in \mathrm{Safety}_B(\pi)$, otherwise $\pi \notin \mathrm{Safety}_B$. We show how various correctness conditions can be modelled as Safety objectives.

Linearizability. Linearizability can be specified as Safety objectives. The important step is to check when each method finishes execution, whether the return value is consistent with some valid serialization of the client. Therefore, the set of states which are unsafe are the ones which have return values of methods which are not consistent with any serialization of the client. For a detailed explanation of checking linearizability as safety, refer to [8] where the reduction is done for methods which operate on a list.

Deadlock freedom. One of the major problems of synchronizing programs using some form of blocking primitives like locks is that deadlocks may arise. This is when two or more threads are waiting for resources held by each other. We can do deadlock detection in a transition graph by checking for the existence of reachable states in which all threads which have not finished execution are waiting for some lock. This can be cast as a Safety objective by adding such states, which do not have any transitions from them, to the set of unsafe states.

Data-race freedom. Data-races occur when two or more threads write to the same shared memory location and one of the writes overwrites the other leading to loss of data. To avoid this we need to check in the game graph model that there are no states where two or more transitions of different threads are enabled and all the transitions write to the same memory location. These kind of states can be marked unsafe and the objective would be to avoid them.

Performance analysis. We will show how to obtain a PI 2-player (resp. PI $2\frac{1}{2}$-player) game with certain quantitative objectives from $\{\!\{\mathcal{P}, C\}\!\}$ (resp. $\{\!\{\mathcal{P}, C, \mathrm{Sch}\}\!\}^P$) such that solution the game is the worst-case (resp. average-case) performance of a correct program allowed by $\mathcal{P}$.

Limit-average and Limit-average safety objectives. The *limit-average* is a quantitative objective that assigns a real-valued number to every play. The limit-average objective consists of a *cost* function $\mathsf{c} : E \to \mathbb{Q} \cup \{\infty\}$, and assigns to a play the long-run average of the weights. Formally, for a play $\pi = (s_0 a_0 s_1 a_1 s_2 \ldots)$, we have $\mathrm{LimAvg}_{\mathsf{c}}(\pi) = \liminf_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n} \mathsf{c}((s_i, a_i, s_{i+1}))$. The *limit-average safety* objectives are a *lexico-graphic* combination of limit-average and saefty objectives: the objective consists of a cost function c, and a set B of bad states, and for a play π, if $\pi \in \mathrm{Safety}_B$, then the value of π is $\mathrm{LimAvg}_{\mathsf{c}}(\pi)$, otherwise it is ∞ (if the safety objective is satisfied, then we have the limit-average value, ∞ otherwise). The limit-average safety objective can be reduced to safety objectives by making the states in B absorbing (sink states with only self-loops) and assigning them weight ∞.

Definition 3.9. *Conisder a sketch $\mathcal{P}$, a program P, a client C, a set I of inputs, a usage model* UsM*, a performance model* PerfM*, and a scheduler* Sch*.*

- *Given an execution e of C on I, its* performance cost *according to* PerfM *is defined to be the total cost of each step of the execution averaged over the number of steps.*
- *The* worst-case performance cost *according to* PerfM *of a client C with inputs I is the supremum of the performance cost of all*

possible executions of C with input I and the corresponding average-case performance cost *is the expectation of performace cost of all possible executions over to the probability distribution over executions when client C is scheduled by* Sch.

- *The worst-case performance cost (resp. average performance cost) according to* PerfM *of a program P used as per the usage model* UsM *is the sum of the worst-case performance (resp. average performance) cost over all inputs and over all clients allowed by the program, averaged over the probabilities of use of various clients.*
- *The worst-case performace cost (resp. average performance cost) of a sketch $\mathcal{P}$ according to a performance model* PerfM*, when used as per the usage model* UsM*, is defined to be the worst-case performance (resp. average-case performance) cost of the best program allowed by $\mathcal{P}$.*

Given a performance model PerfM, i.e., a weighted automata and a transition game graph $\{\!\{\mathcal{P}, C\}\!\}$, we take their product by matching the alphabet of the PerfM with the states of $\{\!\{\mathcal{P}, C\}\!\}$. For example, if there exists a Player 2 transition from state s_1 to state s_2 in $\{\!\{\mathcal{P}, C\}\!\}$ which corresponds to a step in the program where a lock is acquired, there will exist transitions from states (s_1, q_i) to (s_2, q_j) in the product where $\delta(q_i, l) = q_j$, where l is the symbol for lock. The weight of this transition will be the weight of the corresponding transition in PerfM. We denote this product as $\{\!\{\mathcal{P}, C, \mathrm{PerfM}\}\!\}$. The similar product construction for $\{\!\{\mathcal{P}, C, \mathrm{Sch}\}\!\}^P$ is denoted as $\{\!\{\mathcal{P}, C, \mathrm{Sch}, \mathrm{PerfM}\}\!\}^P$. The product constructions give us PI $2\frac{1}{2}$- and PI 2-player game graphs with limit-average safety objectives. We now define the notion of values in games, and in Theorem 3.10 we establish the correspondence of values games and performance costs of sketches.

Values. For a quantitative objective f, the value of two strategies $\tau_1 \in \Gamma_1$ and $\tau_2 \in \Gamma_2$, denoted $Val(f, \tau_1, \tau_2)$, is the expected f value of the unique plays under both τ_1 and τ_2, i.e., $\mathbb{E}^{\tau_1,\tau_2}(f)$. The value of a Player 1 strategy $\tau_1 \in \Gamma_1$, denoted as $Val(f, \tau_1)$, is the maximum value of a play given a counter strategy of Player 2. Formally, $Val(f, \tau_1) = \sup_{\tau_2 \in \Gamma_2} Val(f, \tau_1, \tau_2)$. Similarly, we have $Val(f, \tau_2) = \inf_{\tau_1 \in \Gamma_1} Val(f, \tau_1, \tau_2)$. The value of the game, denote $Val(f)$, is the minimum value that can be guaranteed by Player 1, i.e., $Val(f) = \inf_{\tau_1 \in \Gamma_1} \sup_{\tau_2 \in \Gamma_2} Val(f, \tau_1, \tau_2)$. The memoryless value of the game is obtained as above by restricting the strategies of Player 1 to be memoryless.

Theorem 3.10. *Consider a sketch $\mathcal{P}$, a client C, a performance model* PerfM*, a scheduler* Sch *and a correctness property Φ for $\mathcal{P}$ specified as a safety objective. The memoryless value of the* LimAvg Safety PI 2*-player game $\{\!\{\mathcal{P}, C, \mathrm{PerfM}\}\!\}$ (resp.* PI $2\frac{1}{2}$*-player game $\{\!\{\mathcal{P}, C, \mathrm{Sch}, \mathrm{PerfM}\}\!\}^P$) is equal to the worst-case (resp. average-case) performance cost of the best correct program allowed by $\mathcal{P}$ for the client C (resp. under scheduler* Sch*).*

3.4 Finite-State Abstractions

In the preceding sections, we developed a simple reduction from sketches to PI-games so that the both correctness and performance properties can be checked on the game. However, as mentioned before, it is possible and infact true for most heap modifying programs that the state spaces of the above games are infinite.

In this section, we prove that the correctness and performance properties are preserved under certain abstractions. These results allow us to use abstracted finite versions of the game graphs and transition systems to compute the properties and synthesize from sketches.

Definition 3.11. *A method sketch $\mathcal{P}$ is* analyzable *for* PerfM *if there exists abstraction functions* Abst*,* ch *and* tr *such that the following holds:*

- $\mathrm{Abst}(P, C)$ *is a finite state transition system for every P allowed by $\mathcal{P}$ and every client C,*
- *For all clients C and correctness conditions Φ, a program P allowed by the sketch satisfies Φ when executed as C iff* $\mathrm{Abst}(P, C) \models \mathrm{tr}(\Phi)$*, and*
- *For all clients C, and for all programs P allowed by the sketch,* $[\![P, C, \mathrm{PerfM}]\!] = [\![\mathrm{Abst}(P, C), \mathrm{ch}(\mathrm{PerfM})]\!]$.

Intuitively, ch and tr are functions which transform quantitative and boolean objectives from the real model of the program to objectives on the abstract model. Therefore, the above definition intuitively says that a sketch is analyzable if there exists a finite abstraction on which the correctness and performance costs are preserved.

We show here that any sketch consisting of method sketches as defined in Section 2, i.e., methods that operate on a singly-linked heap with a single traversal of the heap per method, is analyzable. For this, we consider the *lock-step* abstraction introduced in [8]: the reduction was shown to work for verification of safety objectives. We extend the result in two ways: first we show that the abstraction works for the more general problem of games for synthesis, and second it works for the worst-case performance analysis of quantitative objectives.

Theorem 3.12. *Method-automata are analyzable using the order abstraction and lock-step reduction for performance models with parameters locking cost, waiting cost and context-switch costs.*

Proof idea. The correctness part of the proof follows directly from the results of [8]. The key idea in the proof for preservation of performance costs is that under the order abstraction and lock-step reduction, every execution has an equivalent execution in the abstract model for which the locking and waiting costs are preserved and that there exists an execution which has a higher context-switch cost.

Also, the above theorem helps us analyse a large class of other problems. For example, using the order abstraction on finite programs helps us reduce the size of the game graphs so that the analysis and synthesis is much more efficient, for both boolean and quantitative versions.

Small-step big-step optimization. Although, the above theorem helps us construct a finite-state model for many heap-based programs, the state space obtained is still significantly large. Hence even the correctness check (for linearizability) as described in [8] is too slow in practice. We have come up with a very practical optimization, namely *small-step big-step optimization* to reduce the state space by an order of magnitude. For linearizability checking, the lock-step abstraction simulates separate sequential runs of the method expression, and one step of each sequential run is executed for each transition of the finite transition system. Instead we simulate the sequential runs in *big steps*: a collection of sequential steps once in few transitions is combined as a big step. We ensure that the big step happens every time the value of a fresh heap location is chosen. This optimization leads to huge space saving, and as a consequence not only makes the quantitative synthesis feasible but also hugely improves the verification results of [8].

3.5 Computational Complexity of PI Games

We now present the decision problems for PI $2\frac{1}{2}$ and 2-player game graphs, and study the complexity.

Decision problems. Given a quantitative objective f and a rational threshold $q \in \mathbb{Q}$, the decision problem (resp. memoryless decision problem) asks whether there is a strategy (resp. memoryless strategy) τ_1 for Player 1 such that $Val(f, \tau_1) \leq q$.

The classical game theory study always considers the general decision problem which is undecidable for limit-average objectives [13].

Theorem 3.13. *[13] The decision problems for* LimAvg *and* LimAvg-Safety *objectives are undecidable for* PI $2\frac{1}{2}$- *and* PI 2-*player game graphs.*

We now study the complexity of the memoryless decision problems for PI $2\frac{1}{2}$- and PI 2-player game graphs that has not been studied before. We have already shown in Theorem 3.10 that the relevant problem for synthesis from sketches is the memoryless value problem. For the special case of MDPs, the answer to the decision and memoryless decision problems coincide and can be solved in polynomial time [15] (using linear-programming to solve MDPs with safety and limit-average objectives).

Theorem 3.14. *[15] The memoryless decision problem for* LimAvg-Safety *objectives can be solved in polynomial time for MDPs.*

Lemma 3.15. *The memoryless decision problem for* PI 2-*player game graphs with* Safety *and* LimAvg *objectives are NP-hard.*

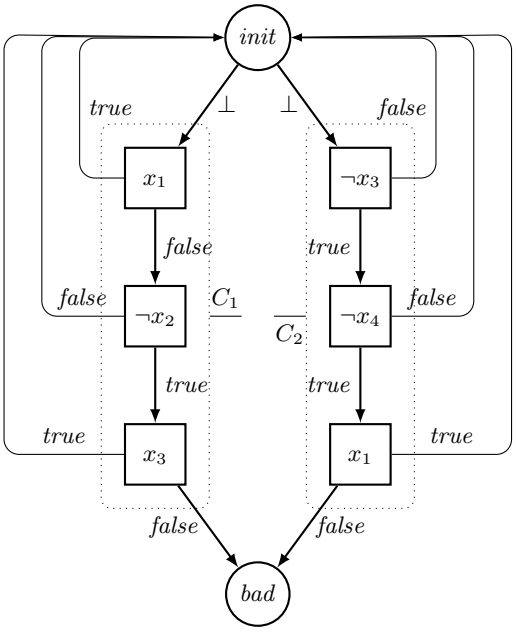

Figure 6. 3-SAT to memoryless partial-information Safety games

Proof. We first show NP-hardness for safety objectives.

(NP-hardness). We will show that the problem is NP-hard by reducing the 3-SAT problem. Given a 3-SAT formula Φ over variables $x_1, x_2, \ldots x_N$, with clauses $C_1, C_2, \ldots C_K$, we construct a partial-information game graph with $N+1$ observations and $3K+2$ states such that Player 1 has a memoryless winning strategy from the initial state if and only if Φ is satisfiable. The construction is described below:

- The states of the game graph are $\{init\} \cup \{s_{i,j} \mid i \in [1,K] \wedge j \in \{1,2,3\}\} \cup \{bad\}$.
- The observations and the observation mapping are as follows: $init$ and bad are mapped with observation 0, and $s_{i,j}$ is mapped with observation k if the j^{th} variable of the C_i clause is x_k or $\neg x_k$.
- $init$ and bad are Player 2 states and all other states are Player 1 states.
- The edges of the game graph are as follows:
  1. For all $i \in [0,K]$, there is an edge from $init$ to $s_{i,1}$ on the symbol $\perp$.
  2. If the j^{th} literal of clause C_i is x_k, there is an edge from $s_{i,j}$ to $init$ on $true$ and to $s_{i+1,j}$ on $false$ (for $j \in \{1,2\}$). For $j = 3$, the edge on $true$ leads to $init$ and the edge on $false$ leads to bad.
  3. If the j^{th} literal of clause C_i is $\neg x_k$, there is an edge from $s_{i,j}$ to $init$ on $false$ and to $s_{i+1,j}$ on $true$ (for $j \in \{1,2\}$). For $j = 3$, the edge on $false$ leads to $init$ and the edge on $true$ leads to bad.
- The objective for Player 1 is to avoid reaching bad and the objective for Player 2 is to reach bad.

Intuitively, Player 2 chooses a clause C_i in the initial state $init$. Player 1 then plays according to her memoryless strategy from each of the states $s_{i,j}$. If the action $a \in \{true, false\}$ chosen in $s_{i,j}$ makes the literal at position j in clause C_i true, control goes back to $init$. Otherwise, the control goes to the next $s_{i,j+1}$. If the choices at all three $s_{i,j}$'s make the corresponding literal false, the control goes to bad. The game graph structure is illustrated in Figure 6.

Given a truth value assignment of x_i's such that Φ is satisfied, we can construct a memoryless strategy of Player 1 which chooses the action at observation i same as the valuation of x_i, and the memoryless strategy is winning for Player 1. In every triple of $s_{i,j}$'s at least one of the edges dictated by this strategy lead to $init$. If that were not the case, the corresponding clause would not have been satisfied. Given a winning memoryless strategy τ_1, the valuation of x_i's which assigns the $\tau_1(i)$ to x_i satisfies each clause C_k in Φ. This follows from a similar argument as above. Hence the hardness result follows.

The above reduction is slightly modified to show that the LimAvg memoryless decision problem is also NP-hard. This can be done by adding a self loop on state bad with weight 1 and attaching the weight 0 to all other edges. Now, Player 1 can obtain a value less than 1 if and only if she has a memoryless winning strategy in the Safety game.

The desired result follows. ■

Lemma 3.16. *The memoryless decision problem for* LimAvg-Safety *objectives for* PI $2\frac{1}{2}$*-player game graphs is in NP.*

Proof. Given a memoryless winning strategy for a Player 1 in a PI $2\frac{1}{2}$-player game graph, the verification problem is equivalent to solving for the same objective on the MDP obtained by fixing the strategy for Player 1. Hence the memoryless strategy is the polynomial witness, and Theorem 3.14 provides the polynomial time verification procedure to prove the desired result. ■

Lemma 3.15 and Lemma 3.16 gives us the following theorem.

Theorem 3.17. *(Complexity). The memoryless decision problems for* Safety, LimAvg, *and* LimAvg-Safety *objectives are NP-complete for* PI $2\frac{1}{2}$- *and* PI 2-*player game graphs.*

4. Synthesis and Evaluation Algorithms

In this section we present algorithms for synthesis of concurrent programs that overcome the theoretical difficulty (of NP-completeness) in cases of practical interest and work efficiently for games from sketches.

Synthesis algorithm. Our quantitative synthesis algorithm (Algorithm 1) takes as input a sequential sketch, a performance measure given as a weighted automaton, a scheduler, a set of clients, and

Algorithm 1 Quantitative synthesis Algorithm

Input: $\mathcal{P}$: sequential data structure sketch
PerfM: performance measure
Sch: a scheduler
Cl: a set of clients
UsM: a usage model
Output: (C_{conc} concurrent data structure) or fail

$L = \text{MemorylessStrategies}(\mathcal{P})$
$L_S \leftarrow \mathsf{LtWtPaCor}(L, \mathcal{P})$
Set ES $\leftarrow \emptyset$
while $L_S \neq \emptyset$ **do**
  Pick $\tau_1 \in L_S$ and remove τ_1 from L_S
  $(\mathsf{correct}, \mathsf{ctrex}) \leftarrow \mathsf{CheckCorrect}(\tau_1, \mathcal{P})$
  if $\neg\mathsf{correct}$ **then**
    $L_S \leftarrow \mathsf{prune}(L_S, \mathsf{ctrex})$
  else
    $\text{val}(\tau_1) \leftarrow \mathsf{EvalPerf}(\mathcal{P}, \text{PerfM}, \text{Sch}, \text{Cl}, \text{UsM}, \tau_1)$
    ES $\leftarrow$ ES $\cup \{(\tau_1, \text{val}(\tau_1))\}$;
if ES $\neq \emptyset$ **then**
  $\tau_1^* \leftarrow \arg\min_{\tau_1}\{\text{val}(\tau_1) \mid (\tau_1, \text{val}(\tau_1)) \in \text{ES}\}$
  $C_{\text{conc}} \leftarrow \text{prog}(\tau_1^*, \mathcal{P})$;
  output C_{conc}
else
  output "Fail."

a usage model and produces as its output a concurrent data structure implementation that is both correct and optimal with respect to the performance measure and the usage model. The algorithm enumerates over the memoryless strategies for player 1 of the PI game graph(MDP) corresponding to the sequential sketch $\mathcal{P}$, and checks for correctness and the performance of the strategies. The correctness and performance evaluation steps are computationally expensive. Our algorithm first uses a *light-weight* pre-processing step based on partial correctness (LtWtPaCor) that efficiently prunes the set of strategies that needs to be checked for correctness, and obtains the list L_S of sensible strategies.The procedure LtWtPaCor uses sanity checks on sequential sketch with respect to synchronization constructs like avoiding unlocking a list node before locking, elimination of duplicate strategies (strategies which are different but equivalent) and considering only the strategies satisfying a particular order of selections in similar choice blocks. In our examples, this method prunes the number of strategies from thousands to less than hundred in less than a second. The strategies in L_S are checked for correctness and performance evaluation in a loop. The first check is the correctness check: if a strategy τ_1 is not correct, then we obtain a witness counter-example ctrex. The counter-example is used to further prune the set L_S of sensible strategies. If a strategy is correct, then we evaluate the performance of the strategy with respect to the performance measure and the usage model. The set ES contains pairs of strategies and their performance values.

We present two algorithms for performance evaluation: one for the worst-case performance, and the other for the average-case performance. Once we obtain the set of strategies that are correct, we choose the strategy that is optimal with respect to performance from ES. The formal description of our synthesis algorithm is given as Algorithm 1. We now describe the performance evaluation algorithms: (a) Algorithm 2 for the worst-case performance and (b) Algorithm 3 for the average-case performance.

Worst case evaluation algorithm. Algorithm 2 is invoked as one of the EvalPerf procedure by Algorithm 1. The algorithm takes as input a sequential sketch, evaluation parameters (as used in Algorithm 1) and a strategy τ_1, and outputs the worst-case performance

Algorithm 2 Worst Case Performance Evaluation

Input: $\mathcal{P}$: sequential data structure sketch
PerfM: performance measure
Sch: a scheduler
Cl: a set of clients
UsM: a usage model
τ_1 : a strategy
Output: $\text{val}(\tau_1)$: performance value of strategy τ_1

for all $C \in \text{Cl}$ **do**
  $\text{Gr} \leftarrow \mathsf{ZC}(\{\!|\mathcal{P}, C, \text{PerfM}|\!\}_{\restriction_{\tau_1}})$
  $\text{val}(\tau_1, C) \leftarrow \mathsf{MaxMeanCycle}(\text{Gr})$
$\text{val}(\tau_1) = \sum_{C \in \text{Cl}} \text{Prob}^{\text{UsM}}(C).\text{val}(\tau_1, C)$

value for the strategy τ_1. The steps of the algorithm are as follows: (a) for each client C, first the algorithm constructs the synchronus product graph Gr of the performance model PerfM (weighted automaton) and the finite state transition system $\{\!|\mathcal{P}, C|\!\}_{\restriction_{\tau_1}}$

(b) then, it finds the maximum mean-value cost of a cycle in this graph (this corresponds to the worst-case performance for the client C over all inputs and all schedules);

(c) finally, it computes the worst-case performance value of the strategy τ_1 as the probabilistic average of these values over the usage model.

The product construction in step (a) has already been described in Section 3. We obtain value of the strategy by computing the maximum mean cycle value in a weighted graph. The two well-known algorithms for this problem are Karp's mean-cycle algorithm with $O(|V| \cdot |E|)$ running time in a graph with $|V|$ vertices and $|E|$ edges, and (b) Howard's strategy improvement algorithm for which the only known bound on running time is exponential. For a summary of various max mean cycle algorithms, see [12]. We tested both approaches, and in our example of large game graphs the strategy improvement algorithm was much faster than the mean-cycle algorithm. For our examples, the mean-cycle algorithm took around a minute on average, whereas the strategy improvement algorithm took less than a second. The strategy improvement algorithm fixes a strategy (one edge per state) in each strongly connected component and then tries to locally improve distance of nodes from the best cycle. When such improvement is no further possible, the strategy improvement stops. For our examples strategy improvement works very well due to the small (bounded) out-degree of each vertex of Gr. The formal description of the worst-case evaluation algorithm is given as Algorithm 2.

Algorithm 3 Average Case Performance Evaluation

Input: $\mathcal{P}$: sequential data structure sketch
PerfM: performance measure
Sch: a scheduler
Cl: a set of clients
UsM: a usage model
τ_1 : a strategy
Output: $\text{val}(\tau_1)$: performance value of strategy τ_1

for all $C \in \text{Cl}$ **do**
  $\text{Gr} \leftarrow \mathsf{ZC}(\{\!|\mathcal{P}, C, \text{Sch}, \text{PerfM}|\!\}^{P}_{\restriction_{\tau_1}}), \text{PerfM}$
  $\text{val}(\tau_1, C) \leftarrow \mathsf{SolveMDP}(\text{Gr})$
$\text{val}(\tau_1) = \sum_{C \in \text{Cl}} \text{Prob}^{\text{UsM}}(C).\text{val}(\tau_1, C)$

Average case evaluation algorithm. Similar to Algorithm 2, Algorithm 3 is also invoked as one of the EvalPerf procedure by Algorithm 1, and hence its inputs coincide with Algorithm 2. The output is the average case performance value for the strategy τ_1.

Like Algorithm 2, the steps of Algorithm 3 are as follows: (a) for each client C, first the algorithm constructs the synchronus product of the performance model PerfM (weighted automaton), the finite state transition system $\{\!\{\mathcal{P}, C\}\!\}_{\restriction_{\tau_1}}$ obtained under τ_1 and the probabilistic scheduler; in contrast to Algorithm 2 the product construction gives us an MDP, not a graph, and this is due to the probabilistic scheduler for the average-case performance; (b) then it finds the limit-average value of the MDP (this corresponds to the average-case performance for the client C for the worst-case inputs and under the probabilistic scheduler); (c) finally, it takes the probabilistic average of these values using the usage model to find the average-case performance value of the strategy τ_1. The product construction in step (a) has already been described in Section 3.

We have implemented the strategy-improvement algorithm for MDPs [14, 15], as they are the most efficient in practice, to obtain the limit-average value of an MDP. The classical strategy-improvement algorithm assumes cost on every state, however, in our model the cost are on edges. The cost of a state is taken as the probabilistic average of the cost of the outgoing edges (the two cost models are equivalent since the Cesaro limit (limit-average frequency) of a state in a Markov chain obtained by fixing a strategy in an MDP is same as the probabilistic average of the Cesaro limit frequencies of the outgoing edges). The strategy improvement algorithm fixes a strategy τ_1 and calculates two vectors v_{τ_1} (value vector) and δ_{τ_1} (deviation vector). The vectors $(v_{\tau_1}, \delta_{\tau_1})$ is the unique (x, y) solution of the following set of linear equations :

$$\left.\begin{array}{l}(a)\ [I - \Delta_{\tau_1}]x = 0;\ (b)\ y + [I - \Delta_{\tau_1}]z = 0;\\ (c)\ x + [I - \Delta_{\tau_1}]y = \mathsf{c}_{\tau_1}.\end{array}\right\} \quad (1)$$

where Δ_{τ_1} is the probability matrix of the Markov chain obtained by fixing the strategy τ_1 in the MDP and c_{τ_1} is the cost vector. Since the graphs we deal with are large (i.e., the probability matrix Δ_{τ_1} is huge but sparse), implementation of methods like Gaussian elimination that require the full matrix to be stored were infeasible. Hence the only choice for implementation are iterative methods which exploit the fact that the matrix is sparse.

The solution given by the equations in (1) was inefficient in practice due to two reasons: (1) if n is the number of states, then the three equations in (1) gives us $3n$ equations in $3n$ variables (x, y, and z each have n component variables), and hence each iteration requires a sparse matrix multiplication of size $3 \cdot n \times 3 \cdot n$; (b) the convergence rate is quite slow for the iterative methods. Our main insight to overcome the first problem is as follows: to retrieve (x, y) from the equations, instead of solving (1) directly, we first solve the following equation:

$$[I - \Delta_{\tau_1}]^3 z + [I - \Delta_{\tau_1}]\mathsf{c}_{\tau_1} = 0 \quad (2)$$

After finding a solution z of (2), we calculate

$$(x, y) = ([I - \Delta_{\tau_1}]^2 z + \mathsf{c}_{\tau_1}, -[I - \Delta_{\tau_1}]z).$$

The major benefit of this two step solution is as follows: the solution of (2) requires 3 sparse multiplications (for $[I - \Delta_{\tau_1}]^3 z$) of a $n \times n$ matrix. This gives us a direct improvement by a factor of 3, and since this step is executed over many iterations our improvement is significant. We note that the final solution of (x, y) after solving (2) is computed only once and not in iterations. Several other iterative methods (like Jacobi, Gauss-Seidel, Biconjugate gradient stabilized etc) that exploit special properties of matrices were infeasible as they did not converge fast on our examples because the matrices did not satisfy the special properties required by these methods. We used the Generalized Minimal Residual (GMRES) method that does not require any special structure of the matrix. The convergence of GMRES can only be theoretically guaranteed in general with some special conditions, however, in all our experiments GMRES converged in a few iterations. Moreover, the GMRES convergence for (2) was much faster as compared to convergence for (1). On average the GMRES method to solve (1) took 10 minutes, where as the solution for (2) was achieved on average in less than 20 seconds. This gives us the computation of the vectors $(v_{\tau_1}, \delta_{\tau_1})$. The improvement step of the strategy improvement algorithm is as follows: it *locally* improves the strategy τ_1 using these two vectors for each state: an action a in state s is better than the current chosen action by τ_1 if either (a) the expected value for a is greater than the current value, i.e., $\sum_{t \in S} \Delta(s, a)(t) \cdot v_{\tau_1}[t] > v_{\tau_1}[s]$; or (b) the expected value for a is the same as the current value and the sum of the cost and expected deviation is greater than the sum of the current value and the current deviation, i.e., $(\sum_{t \in S} \Delta(s, a)(t) \cdot v_{\tau_1}[t] = v_{\tau_1}[s]$ and $\mathsf{c}(s, a) + \sum_{t \in S} \Delta(s, a)(t) \cdot \delta_{\tau_1}[t] > v_{\tau_1}[s] + \delta_{\tau_1}[s])$. The algorithm stops when no improvement can be made. The correctness of the strategy improvement algorithm (i.e., when no further local improvement is possible, then we have a *globally* optimal strategy) can be found in [15]. The formal description of the average-case performance evaluation is given as Algorithm 3.

Counter-example analysis. Counter-example based elimination of synchronization strategies is a method that has been used successfully for concurrent program synthesis [24]. If a synchronization strategy is incorrect, i.e., allows an unsafe execution, we extract a path from the transition game graph which corresponds to the unsafe execution. Now, a projection of this trace on the significant variables (variables which are not used only for synchronization) is taken. We try to simulate this counter-example trace on the rest of the valid strategies for unsafe execution to prune incorrect strategies. As we will see in the examples (Section 6), often many strategies are eliminated with a single counter-example.

5. Cut-off theorem for Linearizability of Lists

We consider certain special properties of concurrent data structure access methods which can ensure linearizability for an unbounded number of threads given that all two-thread clients, each thread running one method, are linearizable. We will show that both our list examples satisfy such properties.

Shared accesses. We model an execution of a method automaton in terms of its accesses to the shared memory (shared variables or heap, denoted by SHM). We classify shared memory accesses into the following categories: reads R, writes W, decision of return value D_R, locking L and unlocking U. The categories distinguish between data reads/writes to shared memory and synchronization accesses via locks. The statement D_R is the last read from the SHM after which the return value of the method can be decided locally. An execution of a client program with k threads is modeled by a sequence over the alphabet $\bigcup_{0<i\leq k}\{R^i, W^i, L^i, U^i, D_R^i\}$, symbols being indexed by thread number.

One-shared-update programs. We define a class of method automata namely *one-shared update* (OSU) method automata. Automata in this class: (i) Perform at most one write W to SHM. (ii) Executes the decision read D_R where the automaton decides the return value and whether it is going to perform W (with a fixed return value).

For the sketch in Figure 7, one can easily verify (also automatically by static analysis) that it gives rise to OSU method automata. For the sketch in Figure 8, the logical remove statement can be taken as the single W statement for the `remove` method due to semantics of lazy marking and retry. For the `remove` method in Figure 7, the write W is issued in the statement `pred.next = curr.next;`. This statement deletes the `curr` element from the list. (Note that this statement is decomposed into two transitions, one that reads the value of `curr.next`, and the other that writes the value of `pred.next`). For the `add` method in Figures 7 and

8, the write W is issued because of the statement `pred.next = node`, i.e. the statement that actually adds a node into the list. Note that the preceding statement `node.next = curr` modifies a node (pointed to by the variable `node`) that is unreachable by other threads. This is an assumption on the inputs of the method. This assignment is therefore not modeled as a write to a SHM location. For the `remove` method in Figure 8 due to semantics of the sketch, we can model the `curr.marked = true` statement as the only shared update because the `add` and `remove` methods wait for the next statement(physical write to the shared list) of the original `remove` method to complete in case it wants to add or remove a node before or after the node being removed, and the `contains` method returns false after the node is marked if it were searching for it. Effect of marking a node in `remove` method is equivalent to removing the node in the same statement for all other methods. Thus, we have Proposition 5.1.

Proposition 5.1. *The programs arising from sketches in Figure 7 and Figure 8 are OSU programs.*

Critical reads. An *action statement* of an OSU method in a given execution e is the write statement W if it executes a write in e or the decision read D_R (last read for deciding the return value) if it does not execute any write statement. We now define critical reads for an OSU method automaton. Intuitively, critical reads for a method automaton are those which determine the result of the action statement.

Definition 5.2. *A set of reads is* critical *for an OSU method iff for any two executions of the method in which the configurations of SHM are the same before the corresponding action statements, the configuration and return value resulting after execution of the action statement are the same.*

Note that the decision read D_R (and hence all its critical reads) are also critical reads for the write statement (W).

Cut-off theorem: basic ideas. Our cut-off theorem will show that for OSU methods a stronger notion of linearizability for all 2 thread-clients, each thread with a single-method, is sufficient (and necessary) to ensure linearizability for all n thread-clients, for all $n \geq 2$. Our proof is by induction on the number of threads. The basic idea of the proof is as follows: let us consider two OSU methods m_1 and m_2 such that they are 2-thread linearizable and an interleaved execution e of m_1 and m_2. Without loss of generality, let the action statement A_1 of m_1 precede the action statement A_2 of m_2 in e. To prove the result for the inductive case we would need the following: ensuring that all reads of m_2 before A_1 can be shifted after A_2, without changing values of the critical reads of m_2. However, this fact cannot be inferred straightforwardly from linearizability. Hence we assume a *stronger* condition (defined as *strong linearizability (SL)*), which strengthens the inductive hypothesis, and then with the stronger inductive hypothesis we show that the stronger inductive claim holds. The other important component of the proof is to preserve synchronization under projection on component threads (longest prefix match of the execution with the language of all allowed executions of the component threads). We show this below.

Given an execution e for a set C_O of OSU methods (including m_1 and m_2) running in parallel with its first action statement being A_1 (corresponding to method m_1), we define *two-method projection* of e w.r.t m_1 and m_2 as the interleaved execution π(denoted by $e|_{\mathsf{m}_1,\mathsf{m}_2}$) which is obtained by first scheduling(as much as possible) statements of both methods in the order in which they occur in e starting with the same configuration of SHM as e, and then if needed completing m_2 as per the configuration of SHM after execution of A_1. Note that π need not match e w.r.t the trace of m_2 but the trace of m_1 will be the same in both π and e.

Preserving synchronization under projection. We claim that the synchronization between the methods is preserved under projection of an execution i.e., given any allowed execution e comprising of n threads each running a single method, any projection e' of e is also an allowed execution. This statement follows for error-free lock-based synchronization since matching lock and unlock statements are always local to a single thread (any other thread unlocking a lock held by this thread results in an error).

Definition 5.3. *(Strong linearizability (SL)) Given a set of k OSU methods $C_O = \{\mathsf{m}_1, ..., \mathsf{m}_k\}$, we say that C_O is n-thread* strongly linearizable(SL)*, if for every interleaved execution e of an n-thread client (each thread running one method from C_O), there is a sequential execution s_e of the same client such that:*

- *In s_e, methods are executed in the order of their action statements in e and the return values of each method in both e and s_e are identical i.e. the execution is serializable in the order of action statements of the OSU methods, and*
- *all critical reads of corresponding methods are identical in both e and s_e.*

Lemma 5.4. *Given two valid executions e_1 and e_2 for a set A of OSU methods running in parallel(including m_1 and m_2) such that (i) A_1 (action statement of m_1) is the first action statement of both the executions and (ii) there is some critical read R_c^2 of m_2 before A_1 in e_1 and the same read R_c^2 is after A_1 but before the next action statement in e_2 then value of the critical read R_c^2 is the same in both e_1 and e_2, provided A is 2-thread strongly linearizable.*

Proof. Consider the *two-method projections* $\pi_1 = e_1|_{\mathsf{m}_1,\mathsf{m}_2}$ and $\pi_2 = e_2|_{\mathsf{m}_1,\mathsf{m}_2}$. By preservation of synchronization under two-method projections, both π_1 and π_2 are valid executions.Now, by 2-thread SL of methods m_1 and m_2, both executions π_1 and π_2 have the same values of all critical reads as the sequential execution $sequential(\mathsf{m}_1, \mathsf{m}_2)$. Hence, the values of the critical read R_c^2 in π_1 and π_2 are the same. Since all statements in e_1 before R_c^2 are only reads, the value of R_c^2 in e_1 and π_1 are the same. Similarly, since R_c^2 is between A_1 and the next action statement in e_2, the configuration of SHM in both e_2 and π_2 at the time of reading R_c^2 can change only due to A_1 for which all critical reads are identical in e_2 and π_2 (reads on the initial configuration of the SHM), hence, value of R_c^2 in both executions e_2 and π_2 are identical. This proves that value of the critical read R_c^2 is same in both e_1 and e_2. ■

Theorem 5.5. *(Cut-off theorem). Given a set of k OSU methods C_O, if C_O is two-thread strongly linearizable, then C_O is n-thread strongly linearizable for all $n \geq 2$.*

Proof. The proof is by induction on n.

Base case $n = 2$ is true by assumption. We assume $n-1$ thread strong linearizability. Now, we consider n-threads each running a method from A (say $\{\mathsf{m}_i : 1 \leq i \leq n\}$ and an interleaved execution e for these n methods. Let the order of action statements in e of the methods be $A_1, A_2, ..., A_n$. We consider all critical reads before A_1 for method m_i and shift all statements of m_i before A_1 to just after A_1. Such an execution is definitely allowed since locking and unlocking is thread-specific(at any instant locks are associated with the thread holding it and only that thread's statement can release the lock), any thread can start executing at any point during the execution of other methods and moreover all return values and list configuration remains the same because there is only one action statement A_1 that can change critical reads of m_i but from lemma 5.4 and two-thread strong linearizability, all critical reads are maintained. We perform this step for all methods having a statement before A_1 in e and get an execution e' with method m_1 being separated from rest of the methods without affecting any return values or critical reads. Now, by $(n-1)$-thread

linearizability, methods $\{\mathsf{m}_2, ..., \mathsf{m}_n\}$ can be strongly linearized as $sequential(\mathsf{m}_2, ..., \mathsf{m}_n)$. Hence e is equivalent to a n-method sequential execution which satisfies all conditions of strong linearizability. ■

Strong linearizability for list examples. In our list examples, there is no critical read for any method m if it does not execute a W (only read is D_R itself for deciding return value) and there is exactly one critical read statement R_C just before the write statement W. Consider two methods m_1 and m_2(from sketch) and an allowed interleaved execution e: if a method does not execute a W then SL follows from the fact that there is only one read D_R for deciding the return value for that method.

The sequential execution in order of the action statements will definitely preserve all critical reads and return value as there is at most one write statement that can change the list. If both methods perform a write and the value of R_C for m_2 changes when it executes after action statement of $\mathsf{m}_1(A_1)$ in $sequential(\mathsf{m}_1, \mathsf{m}_2)$, then the execution is not linearizable (R_C is either successor assignment (`remove`) or decision of next list node (`add`), which if read wrongly will result into different configuration of the list).

Lemma 5.6. *For our list examples, two-thread linearizability implies two-thread SL.*

Hence we have the following implication chain: two-thread linearizability implies two-thread SL (by Lemma 5.8), two-thread SL implies n-thread SL (by Theorem 5.5), and n-thread SL implies n-thread linearizability since SL is a stronger condition than linearizability. We have the following corollary.

Corollary 5.7. *For our list example sketches in Figure 7 and Figure 8, if* `add, remove,contains` *are two-thread linearizable, then they are n-thread linearizable for all $n \geq 2$.*

Strong linearizability for list examples. In our list examples, there is no critical read for a method m if it does not execute a write statement (only read is the decision statement itself) and there is exactly one critical read statement R_c just before the write statement W in every method that executes a W. Two thread linearizability ensures the following: Consider two methods m_1 and m_2 and an allowed interleaved execution e:

- if one of them does not execute W, then they are SL because one method does not change the SHM at all and decides the return value in just one read.
- if both methods perform a write, then suppose the value of the critical read for m_2 changes when it executes after A_1 (action statement of m_1), then the execution (although allowed) is not linearizable (because in our list examples, the critical read is either successor assignment (remove) or decision of next node (add), which if read wrongly will definitely result into different configuration of the SHM (list)).

Lemma 5.8. *For our list example, two-thread linearizability implies two-thread SL.*

Hence we have the following implication chain: two-thread linearizability implies two-thread SL (by Lemma 5.8), two-thread SL implies n-thread SL (by Theorem 5.5), and n-thread SL implies n-thread linearizability since SL is a stronger condition than linearizability. We have the following corollary.

Corollary 5.9. *For our list examples sketches in Figure 7 and Figure 8, if* `add, remove,contains` *are two-thread linearizable, then they are n-thread linearizable for all $n \geq 2$.*

```
choice C1 : { lock.lock(); pred.lock();
  curr.lock(); lock.unlock(); pred.unlock();
  curr.unlock(); skip; }
public (Node, Node) find(Node head, Node item)
    Node pred = head;
    choice C1;  //should be pred or lock.lock()
    Node curr = pred.next;
    choice C1; //should be curr.lock()
    while (curr.key < item.key)
        choice C1; //should be pred.unlock()
        pred = curr; curr = curr.next;
        choice C1; //should be curr.lock()
    return (pred,curr);
public boolean remove(Node item)
    (pred, curr) = find(head, item);
    if (curr.key == node.key)
        pred.next = curr.next; ret = true;
    else ret = false;
    choice C1; //should be pred.unlock()
    choice C1; //should be curr.unlock() or lock.unlock()
    return ret;
```

Figure 7. Coarse-vs-fine sketch

6. Experimental Evaluation

We have implemented the algorithms presented in Section 4 in a tool as follows: The input is a Java file containing the sequential sketch with choice blocks. The methods are parsed into method automata and strategies are eliminated by sanity checks, simple assertion checking with SPIN model checker [19], and finally using counter-example based analysis. We obtain weighted transition systems (or MDPs) from the the performance model (and scheduler) for the correct strategies. For the analysis of weighted transition systems we use the implementation of Howard's policy iteration in the LEMON library [2] and for average case analysis on MDPs, we use classical policy iteration with the novel optimizations explained in Section 4. We use SPARSELIB library for sparse matrix operations and GMRES method [4].

Producer-Consumer Example. We implemented the synthesis model for the producer-consumer example, as discussed in Section 1. Candidate strategies were reduced from $7^6 = 117649$ to 19 using LtWtPaCor procedure and we were able to synthesize the two correct strategies S_2 and S_3 (with 9 more equivalent strategies). We evaluated the strategies for worst case schedule and inputs on 3 threads with the performance model P_2 having only locking cost. The value for strategy S_2 was much better than that for S_3 as expected since S_2 uses lesser number of locks in every execution.

Coarse-grained vs Fine-grained Locking. Consider the sketch in Figure 7. The sketch (depicting `remove` method) provides six choice blocks for synchronization between various methods (using locks). The `contains` and `add` methods use `find` subroutine similar to `remove` method to perform their respective actions over the shared list data structure implementing a set. The sketch encompasses two well known strategies for lock based synchronization: Fine-Grained and Coarse-Grained locking. We find the optimal strategy using worst-case analysis with various parameters in the performance and usage models using Algorithm 1 and Algorithm 2.

Parameters. We use a single state performance automaton with locking cost (denoted by l_c) and waiting cost (denoted by w_c), a usage model parameterized by three values: probabilities that govern how often `contains`, `add` , `remove` are called (denoted by the probabilities $(p_c{:}p_a{:}p_r)$).We consider 6 clients (all 2-method parallel clients). As shown in section 5, these clients are sufficient to guarantee linearizability for all possible clients (on any number of threads).

The sketch in Figure 7 also gives rise to 117649 different list data structure implementations i.e., the synchronizer has these many different memoryless strategies in the corresponding game. First, the static analysis (LtWtPaCor) is applied. Then, we per-

UsM	Costs(l_c,w_c)	CG	FG	FG'	CF
$u1$(90:9:1)	$(2,1)$	1.55	2.27	2.28	2.43
$u1$(90:9:1)	$(1,1)$	1.05	1.27	1.29	1.56
$u2$(50:45:5)	$(1,1)$	1.17	1.27	1.31	1.75
$u3$(1:1:1)	$(1,1)$	1.31	1.27	1.28	1.78
$u3$(1:1:1)	$(1,2)$	2.05	1.62	1.63	2.62

UsM	$(5,1)$	$(2,1)$	$(1,1)$	$(1,2)$	$(1,5)$
u1(90:9:1)	CG	CG	CG	FG	FG
u2(50:45:5)	CG	CG	CG	FG	FG
u3(1:1:1)	CG	CG	FG	FG	FG

Table 1. Perf. costs and Best strategies for CG vs FG locks

form a stricter partial correctness check with production of the explicit state graph for the candidate strategy (using SPIN model checker [19]). Then, we perform a complete correctness (linearizability) check which produces a counter-example in case the strategy is incorrect. We use this counter-example to efficiently prune the list of strategies. Note that we use the improved linearizability check from Section 3.4 for the complete correctness check. The correctness check without these improvements are too expensive to repeat for each strategy.

The initial static analysis (LtWtPaCor) takes less than a second. The number of strategies that are left (i.e. $|L_S|$) is only 72. We then perform the stricter partial correctness check which took around 5 to 10 seconds on each strategy and eliminates 26 strategies. We then perform correctness checks with counter-example based elimination on the remaining strategies. A complete correctness check takes around a minute for each strategy. However, counter-example based elimination is almost instantaneous and the first and second counter-examples eliminate 25 and a further 15 strategies respectively. We then obtain the 4 correct strategies for which we perform the worst case analysis. These 4 strategies are evaluated with the performance model and a usage model using worst case analysis, as described in Algorithm 2. The evaluation for all 6 (two-method) clients takes around 2 to 5 seconds per strategy. The total time for synthesis is around 11 minutes.

We denote the four correct strategies as FG, CG, FG', and CF. The FG (Fine-Grained) and CG(Coarse-Grained) strategies correspond to standard synchronization methods via locking. The FG' strategy is similar to FG except for the fact that it unlocks the `curr` and `pred` pointers in the reverse order and hence, delaying the other thread longer in some runs. CF strategy uses both fine and coarse grained locks and is always more expensive.

We consider three usage models (`Contains`:`Add`:`Remove`) $u1$(90:9:1), $u2$(50:45:5), and $u3$(1:1:1) (as suggested in [17]) and some performance models with various locking and waiting costs (l_c,w_c). The best strategy for each pair of input parameters and the performance costs of each correct strategy for some cost models are summarized in Table 1.

In conclusion, we were able to synthesize both fine-grained and coarse-grained locking strategies and show that performance of strategies varies with both performance cost parameters as well as the usage model. Although FG strategy provides opportunity for more concurrency, the cost of invoking locking and unlocking routines can make it perform worse than CG locking strategy. These results demonstrate the flexibility of quantitative game models for performance-based synthesis of concurrent programs.

Lazy-List Synthesis. Consider the sketch in Figure 8 which encompass three different strategies for lock and marking based synchronization.The choice block description and the `find` subroutine is the similar to the previous example except that the `find` subroutine is now lock-free. The `add` method has the same structure as the `remove` method except that it does not mark any nodes of

UsM	$(1,0,0)$	$(5,1,1)$	$(0,1,0)$	$(1,1,5)$	$(5,3,2)$
u1	$g_{\mathrm{CG}}/l_{\mathrm{CG}}$	g_{CG}	$l_{\mathrm{CG}}/l_{\mathrm{FG}}$	g_{CG}	l_{CG}
u2	$g_{\mathrm{CG}}/l_{\mathrm{CG}}$	l_{CG}	$l_{\mathrm{CG}}/l_{\mathrm{FG}}$	g_{CG}	l_{CG}
u3	$g_{\mathrm{CG}}/l_{\mathrm{CG}}$	l_{CG}	$l_{\mathrm{CG}}/l_{\mathrm{FG}}$	g_{CG}	l_{CG}

UsM	Costs(l_c,w_c,s_c)	l_{FG}	l_{CG}	g_{CG}
$u1$(90:9:1)	$(5,1,1)$	1.83	1.29	1.24
$u1$(90:9:1)	$(5,3,2)$	2.57	2.01	2.99
$u2$(50:45:5)	$(5,1,1)$	1.35	0.98	1.03
$u2$(50:45:5)	$(1,1,5)$	2.62	2.62	1.56
$u3$(1:1:1)	$(5,1,1)$	1.31	1.27	1.28
$u3$(1:1:1)	$(0,1,0)$	0.10	0.10	0.89
$u3$(1:1:1)	$(1,1,1)$	0.71	0.63	0.95

Table 2. The best strategies for Lazy-List sketch

the shared list. The `contains` method is lock-free but checks for marks for deciding presence or absence of a node.

Note that we do not synthesize the marking strategy, and the expression `!pred.marked && !curr.marked && pred.next == curr` is hard-coded into the sketch as it does not affect the performance analysis (which we are demonstrating in this example).

The three main strategies allowed by this sketch are delayed Fine-Grained Lazy (commonly Lazy synchronization) (l_{FG}), global Coarse-Grained Locking (g_{CG}) and delayed Coarse-Grained Lazy (l_{CG}) synchronization which uses a global lock after traversing the list. We find the optimal strategy for various performance and usage model parameters using average case analysis in Algorithm 1.

Parameters. The performance model we consider is similar to the one from the previous case study. However, it includes a cost s_c for each context switch. We perform average-case analysis in this case study with a uniform scheduler (shown in Figure 5). We use the same clients and usage models as in the previous example.

The sketch in Figure 8 gives rise to 117649 different implementations. As in the previous case the initial static analysis (LtWtPaCor) reduced the number of strategies to a few (23) and counter-example based elimination removes the rest of the wrong strategies using just 2 counter-examples. We then obtain the 6 correct strategies. We perform the average case analysis for these strategies. The evaluation for all 6 (two-method) clients takes around 2 minutes per strategy. The overall time for synthesis is around 20 minutes. We summarize the results for synthesis and present the values for the three optimal strategies in Table 6.

We observe that for performance cost values like $(5,3,2)$ and $(1,1,1)$, the best strategy is the hybrid l_{CG} strategy which is achieves a fine balance between the locks used and idling time. We implemented the hybrid l_{CG} strategy in Java and observed that it performed favorably to the others on a desktop computer under some conditions. The results again demonstrate the flexibility of quantitative game framework for synthesis of concurrent programs.

```
public boolean remove(Node item) {
    choice C1; //should be head.lock() or skip
    (pred, curr) = find(head, item);
    while(true)
        choice C1; //should be pred.lock();
        choice C1; //should be curr.lock();
        if(!pred.marked && !curr.marked && pred.next == curr)
            if (curr.key == node.key)
                curr.marked = true;   //logically remove
                pred.next = curr.next;   //physically remove
                ret = true;
            else
                ret = false;
        choice C1; //should be pred.unlock()
        choice C1; //should be curr.unlock()
    choice C1;//should be head.unlock() or skip
    return ret;
```

Figure 8. Lazy-List sketch

References

[1] Full version. http://pub.ist.ac.at/ cernyp/popl11synth.pdf.

[2] Lemon graph library. http://lemon.cs.elte.hu/trac/lemon.

[3] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, 2007.

[4] E. Bertolazzi. Sparselib: A c++ class library for big vector and sparse matrices, 1999.

[5] R. Bloem, K. Chatterjee, T. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In *CAV*, 2009.

[6] S. Burckhardt, R. Alur, and M. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.

[7] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: a complete and automatic linearizability checker. In *PLDI*, 2010.

[8] P. Černý, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur. Model checking of linearizability of concurrent list implementations. In *CAV*, 2010.

[9] K. Chatterjee, T. A. Henzinger, B. Jobstmann, and R. Singh. Measuring and synthesizing systems in probabilistic environments. In *CAV*, 2010.

[10] A. Church. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians*, 1962.

[11] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, 1981.

[12] A. Dasdan and R. K. Gupta. Faster maximum and minimum mean cycle algorithms for system performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17, 1997.

[13] A. Degorre, L. Doyen, R. Gentilini, J.-F. Raskin, and S. Toruńczyk. Energy and mean-payoff games with imperfect information. In *CSL*, 2010. To appear.

[14] E. Feinberg and A. Shwartz, editors. *Handbook of Markov Decision Processes - Methods and Applications*. 2002.

[15] J. Filar and K. Vrieze. *Competitive Markov decision processes*. 1996.

[16] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, 2001.

[17] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, 2005.

[18] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12, 1990.

[19] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[20] Y. Liu, W. Chen, Y. Liu, and J. Sun. Model checking linearizability via refinement. In *FM*, 2009.

[21] M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical report, U. of Rochester, 1995.

[22] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, 1989.

[23] M. Segalov, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Abstract transformers for thread correlation analysis. In *APLAS*, 2009.

[24] A. Solar-Lezama, C. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI*, 2008.

[25] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[26] V. Vafeiadis. Automatically proving linearizability. In *CAV*, 2010.

[27] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.