Composable Specifications for Structured Shared-Memory Communication

Benjamin P. Wood   Adrian Sampson   Luis Ceze   Dan Grossman

University of Washington

{bpw,asampson,luisceze,djg}@cs.washington.edu

Abstract

In this paper we propose a *communication-centric* approach to specifying and checking how multithreaded programs use shared memory to perform inter-thread communication. Our approach complements past efforts for improving the safety of multithreaded programs such as race detection and atomicity checking. Unlike prior work, we focus on what pieces of *code* are allowed to communicate with one another, as opposed to declaring what data items are shared or what code blocks should be atomic. We develop a language that supports composable specifications at multiple levels of abstraction and that allows libraries to specify whether or not shared-memory communication is exposed to clients. The precise meaning of a specification is given with a formal semantics we present. We have developed a dynamic-analysis tool for Java that observes program execution to see if it obeys a specification. We report results for using the tool on several benchmark programs to which we added specifications, concluding that our approach matches the modular structure of multithreaded applications and that our tool is performant enough for use in development and testing.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—*reliability*; D.2.5 [*Software Engineering*]: Testing and Debugging—*monitors, testing tools*; D.3.2 [*Programming Languages*]: Language Classifications—*Concurrent, distributed, and parallel languages*; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about programs—*specification techniques*

General Terms Languages, Verification, Reliability

Keywords concurrency, software reliability, bug detection, annotation, specification, shared memory

1. Introduction

With the move to multicore architectures, more and more applications are being written with multiple threads that communicate via shared memory. While many high-level programming languages, Java being a canonical example, provide built-in support for shared memory, developers still struggle to build robust and effective multithreaded programs. On the one hand, shared memory provides a simple abstraction because inter-thread communication is implicit, so programmers need not move data explicitly. On the other hand, a key reason why shared-memory programs are so difficult to write and understand is precisely that inter-thread communication—or its absence—is implicit. The vast amount of programming-languages research on static and dynamic analyses to detect programming errors such as data races, deadlocks, and atomicity violations has helped to address this problem.

In this work, we provide a new and complementary approach to specifying and checking multithreaded safety properties. We believe our specifications more directly match the code structure of programs. Prior work has focused either on *data invariants*, such as object o_1 is thread-local or object o_2 is always protected by lock l, or on *isolation invariants*, such as statement s appears to execute atomically. We focus on *code-communication invariants*, such as if method m_1 writes to memory, another thread will read that write only when executing m_2 or m_3. For example, a partial specification for a queue library could state that writes by `enqueue` should be read only by `enqueue` (for the queue size) or `dequeue` (for the size and the data). To keep specifications simple, we do not describe *what data* is communicated between threads, only *which methods* communicate between threads. In this way, our new communication-centric approach is complementary to existing approaches. In a sense, this captures some of the explicitness of a message-passing model while preserving the conveniences of shared memory.

An analysis tool can then check these specifications. In this work, we develop a dynamic-analysis (i.e., debugging) tool to determine if a program execution violates the program's specification. For the example above, this checking would ensure that the queue abstraction is not violated via unexpected inter-thread communication.

A naive implementation of this approach to specification and checking would work as follows: For every method m, have developers list every method m' that can read data written by m when m and m' run in different threads. The dynamic analysis would then record the relevant metadata with each write and check it for each read. There are several reasons this description is naive, and overcoming these challenges is the primary research contribution in our design:

- *Method calls:* Methods often use callees to perform the actual writes and reads of shared memory. We use a notion of "inlining" in our specifications to indicate that memory accesses are performed on behalf of the caller.
- *Conciseness:* If many methods all communicate with each other, the specifications could suffer a quadratic blow-up. A notion of communication "groups" avoids this problem.
- *Local specifications:* Modern applications are too large for anyone to have a single global view of all inter-thread communication. We design method annotations that need to describe communication only within a conceptual module boundary.
- *Layered communication:* Specifications must capture the intuition that communication can be described at multiple levels of abstraction. For example, a `producer` may pass data to a `consumer` via a queue library. We can specify and check communication at the producer/consumer level and the queue level simultaneously. Naturally, our approach supports an arbitrary number of layers.
- *Encapsulated communication:* Libraries often perform communication that is abstracted away from clients. For example, inside a queue library `dequeue` communicates to `enqueue` (via the queue-size field), but the specifications used to check clients of the library should not consider this communication. We define "communication modules" and "interface groups" to address this essential abstraction issue.

Overall, the specification language has several essential, subtle, and synergistic features. We motivate these features with canonical examples and define them precisely with a formal semantics. This semantics formally describes when a dynamic memory operation violates the specification of allowed communication.

We have also implemented a real dynamic checker that processes specifications written as Java annotations and uses Java bytecode-instrumentation to perform checking. As will be clear after describing our specification language, checking in the general case requires storing the call-stack with each memory write and comparing it to the call-stack at each memory read. By storing only the portion of the call-stack relevant to a program's specifications and aggressively using memoization for everything related to call-stack checking, our tool can run programs with an overhead of approximately 5–10x for most programs—too slow for deployed software, but acceptable for a debugging tool. We describe how to use the tool to identify a program's communication and how to use it interactively to help develop specifications for "legacy" (already written, but unspecified) programs.

We have evaluated our tool by annotating small programs from the Java Grande benchmark suite as well as three large applications from the DaCapo suite. We measure the precision and conciseness of annotations as well as our tool's performance. We conclude that specifying legacy applications is difficult but informative, and we believe that co-developing new applications and their specifications will be even more helpful.

In summary, our contributions are:

- A new communication-centric approach to specifying shared-memory communication
- A specification language that naturally supports modularity and shared-memory communication at multiple layers of abstraction
- A formal semantics for our language
- A dynamic checker that is performant as a debugging tool
- An evaluation of our language and checker on benchmark programs

The rest of this paper proceeds as follows. The next section presents an example to motivate our communication-centric approach and to distinguish it from other approaches. Section 3 describes our specifications and additional examples. Section 4 provides a formal semantics of specification checking to remove any ambiguity regarding the meaning of our modular specifications. Section 5 describes our dynamic-analysis tool for Java. Section 6 evaluates our tool on benchmark applications we annotated. Finally, Sections 7, 8, and 9 discuss future work, related work, and conclusions, respectively.

2. Illustrative Example

In this section, we use a short example to give a basic sense of our communication-centric specifications and how they complement other approaches. This section does not present the full specification language nor does it provide precise definitions.

The Code. The code skeleton for our example appears in Figure 1. It depicts a hypothetical image-rendering application where we imagine the `render` method was recently changed to parallelize the application so that each of `nthreads` threads now processes an equal fraction of the image's pixel rows. (Calls to `render` by each thread are not shown.) We further assume a separate thread initiates the rendering and then calls `getImage` to obtain the result. The

```
@Group("Image")
public class Renderer {
  volatile int curLine;
  final int totalLines;
  ConcurrentMap<Integer,Pixel[]> outputImage;

  // Render line (nthreads * n + tid) for every n.
  @Writer({"Image"})
  void render(int tid, int nthreads) {
    for (curLine = tid;
         curLine < totalLines;
         curLine += nthreads)
      outputImage.put(curLine, expensiveCall(...));
  }

  // Return after rendering is finished
  @Reader({"Image"})
  ConcurrentMap<Integer,Pixel[]> getImage() {
    while (curLine < totalLines) /*spin*/ ;
    return outputImage;
  }
}
```

Figure 1. A simple concurrency bug that can be caught using code-centric communication specification.

getImage method observes the curLine field in order to wait for rendering to finish.

Unfortunately, the naive parallelization of the code introduced a bug: curLine is a field now shared among the calls to render, leading to potentially wrong output and loop conditions.

Our Specifications. Figure 1 also includes the method annotations we use to specify the allowed inter-thread communication. Overall, these specifications indicate that getImage can read memory written by render. We place no restrictions on intra-thread communication; we always mean that the read takes place in a different thread from the write. Because the specifications do not allow render to read memory written by render, any execution that called render from multiple threads would violate the specification (via curLine) and our dynamic analysis would report an error.

Because our specification language was designed for larger programs, annotations use several features that are needed less for tiny examples. First, rather than specify directly the write-to-read communication from render to getImage, we define a *communication group* (Image) and specify that render is a write-member of the group (but crucially not a read-member) and getImage is a read-member. Second, the group itself is (implicitly) private, meaning the communication it specifies is not propagated to callers outside of the module (given our Java substrate, we by default equate packages with modules for our purposes). That way, a caller of a method like createScene (not shown) need not be aware that the callee is using concurrency.

The use of a ConcurrentMap in our example provides another motivating example for distinguishing communication internal to a module from external communication. Internally, concurrent calls to put do potentially communicate (e.g., if the keys are the same, the later call must detect this by reading shared memory internal to the data structure and overwrite the first mapping). But external clients should "see" shared-memory communication only from put to get (and similar methods). Modular specifications for ConcurrentMap would make this distinction (see Sections 3.4 and 3.5); no additional annotations are needed for callers outside the library.

Other Approaches. In many cases, our specifications may detect the same concurrency errors as other approaches such as race detectors and atomicity checkers, which is interesting in and of itself since what we are specifying is fundamentally different. In other cases, the errors detected are complementary. In fact, for our example, these approaches are unlikely to identify the problem:

- Race detectors: The program has no data race. The programmer correctly declared curLine as volatile to allow the asynchronous polling in getImage. Unfortunately, this masks the higher level race in render. The accesses to the concurrent map are properly synchronized by the library implementation.
- Atomicity checkers: Calls to render are not necessarily atomic, but they are *also* not atomic in a *correct* version of the code. Since each call to render makes several calls to outputImage.put, mutating the concurrent map, these put operations may interleave when multiple threads run render concurrently, meaning render is not strictly atomic. render is only atomic at a higher level of abstraction that accounts for the fact that calls to outputImage.put with distinct keys commute with each other. Therefore, an atomicity checker reporting that render is not atomic is not helpful; it does not distinguish correct from incorrect code.

3. Communication Specifications

In this section we present the fundamental concepts of inter-thread communication we use and how to specify them, revising our definitions as we introduce each new concept. Section 3.1 first gives a simple definition of what it means for one method to communicate to another, in terms of dynamic memory operations. It then considers naive approaches for specifying all possible communication. We use these observations to motivate *communication inlining* (Section 3.2) and *communication groups* (Section 3.3). These two concepts suffice for annotating small programs, but larger programs benefit from modular specifications, using features presented in Sections 3.4 and 3.5. Table 1 summarizes the main concepts our specifications embody.

Concept	Purpose	Section
Communication inlining	Methods that perform communication solely on behalf of their callers are allowed to communicate whenever their callers are.	3.2
Communication groups	Concisely specify many related communicating method pairs.	3.3
Communication modules	Build communication abstractions to avoid whole-program specifications.	3.4
Stack segments	Enforce communication abstractions by partitioning each communicating call stack to isolate communication in distinct modules.	3.4
Interface groups	Encapsulate or expose communication at module boundaries.	3.5

Table 1. Overview of communication specification concepts presented in this paper.

3.1 Method Communication

We consider *inter-thread communication* only. If thread t_w executes a memory write operation in the dynamic scope of a call to some method m_w and the result of this operation is later read by a memory read operation executed in the dynamic scope of a call to method m_r by a different thread t_r, then we say that m_w communicates to m_r. (The same principles apply to synchronization: when a thread acquires a lock last released by another thread, communication occurs.) Generalizing to nested method calls, it is clear that every method on the call stack of t_w at the time of its write operation communicates to every method on the call stack of t_r at the time of its read operation.

A key insight in this definition is that the communication effects of memory and synchronization operations are dynamically—not statically—scoped. A method m may communicate through a memory operation in m or any transitive callee of m. This distinction captures a common idiom that is not specific to shared-memory programs: many methods execute memory operations on the behalf of others. (We will exploit this relationship further to develop modular communication abstractions in Section 3.4.)

For example, consider the simple vector implementation outlined in Figure 2. Clients are responsible for synchronizing access to the vector. The add method calls `Util.expand` to expand the underlying array if it is already full when trying to add a new item. Assume one thread calls add and triggers an expansion with expand, which reads all the items in the current array and writes them into a new larger array before add writes an item into the new array. Next, if another thread calls `get`, requesting a different index than that of the newly added item, it reads an element in the array, reading the result of a memory write operation executed in expand. This single write-read pair causes both expand *and* add to communicate to `get`, since the write operation executed in the dynamic scope of both methods.

Naive Approaches to Specification. Two naive ways to specify communication are immediately obvious from our definition of communication. The first is to specify every pair of call stacks that is allowed to communicate. Although this approach gives fully context-sensitive precision to specifications, it would be combinatorial in size and would yield a whole-program specification, clearly a non-starter. The second approach is to enumerate all pairs of methods that are allowed to communicate. A memory read operation is valid under such a specification when for all methods m_w on the call stack at the last write to its target and all methods m_r on the call stack at the read, (m_w, m_r) is in the specification. While this approach is less expensive than enumerating pairs of call stacks, it yields whole-program specifications that are still quadratic in size. In the remainder of this section, we

```
@Group("Vector")
class SimpleVector {
  Item[] elements = new Item[10];
  int size = 0;

  @Writer({"Vector"})  @Reader({"Vector"})
  void add(Item item) {
    if (size == elements.length)
      elements = Util.expand(elements);
    elements[size++] = item;
  }
  @Reader({"Vector"})
  Item get(int i) {
    return elements[i];
  }
  @Writer({"Vector"})  @Reader({"Vector"})
  Item replace(int i, Item item) {
    Item old = elements[i];
    elements[i] = item;
    return old;
  }
  ...
}

class Util {
  static Item[] expand(Item[] array) {
    Item[] tmp = new Item[array.length * 2];
    for (int i = 0; i < array.length; i++)
      tmp[i] = array[i];
    return tmp;
  }
}
```

Figure 2. A simple vector implementation.

harness several important observations on program and communication structure to implement specifications that overcome the limitations of these naive specifications.

3.2 Communication Inlining

Many methods communicate only incidentally, when their callers use them to operate on shared data. For example, there is nothing meaningful about communication performed by `Util.expand` (introduced in Section 3.1 and shown in Figure 2) except in the context of its callers. While it is obvious that a useful specification must *allow* communication between `expand` and the vector `get` method, a specification that explicitly *declares* communication between `expand` and `get` is misleading and unwieldy outside the vector implementation. The specification would likely need pairs containing `expand` and many other methods.

A better way to understand communication in `expand` is that `expand` is allowed to communicate whenever its caller is. We refer to this as *communication inlining*. Communication due to memory operations in `expand` is simply treated as though `expand` is inlined into its caller.

Methods that communicate only on behalf of their callers are so prevalent that all methods are communication-inlined by default unless the specification places explicit restrictions on their communication. In practice, this convention alone has a significant simplifying impact on the complexity of a communication specification. In the vector implementation, for example, the naive pairwise specification requires that `expand` communicate nearly everywhere `add` does. Leaving `expand` inlined removes the need for all these extra pairs.

In the remainder of this paper, when we refer to a stack or a call stack, we mean the version with all inlined methods removed. Inlining yields an interesting property of specifications: the specification in which all methods are inlined allows all communication in a program. Since the call stack is conceptually empty at every communicating write and read operation, the set of methods that communicate as a result is empty. This property becomes particularly useful for developing specifications incrementally once we introduce more modular specification features in Section 3.4.

3.3 Communication Groups

Programs often contain sets of methods where all or nearly all pairs of methods within the set communicate. The vector implementation in Figure 2 is a prime example of this pattern. The three methods shown, in addition to others that are omitted (e.g., `contains`, `find`, and `remove`), communicate with each other (through the `elements` array and the `size` field). For a set of n related methods like this, a naive pairwise specification would include $O(n^2)$ annotations.

The Communication Group Primitive. The *communication group* is the basic unit of a communication specification, and serves to express many communicating pairs in a set of related methods concisely. A group $G = (W, R)$ is a pair of the set W of the group's *writer* methods and the set R of its *reader* methods, representing the set of pairs in the cross product $W \times R$. The writers and readers of a group are collectively referred to as its *members*. Separating the two types of members facilitates the expression of common patterns where certain methods should read values written by others in the group but should not write values that the others may read (or vice versa).

The `@Group`, `@Writer`, and `@Reader` annotations in the vector implementation are the Java annotation equivalent of the following group:

$$G_{Vector} = (\{\texttt{add}, \texttt{replace}\}, \{\texttt{add}, \texttt{get}, \texttt{replace}\})$$

The `get` method is a reader but not a writer in this group. For convenience reasons, our Java specifications use decentralized notation: `@Group("Vector")` declares a group by name; each method is annotated as a writer or reader in zero or more groups:

```
@Writer({"Vector"}) @Reader({"Vector"})
```

Checking the communication resulting from a memory read operation against a specification is simple to define:

Definition 1 (Valid Simple Communication)**.** *A read of x is always valid if x was last written in the same thread. If x was last written by a different thread t_w, then it is valid for a thread t_r to read x if for all methods m_w that were on t_w's call stack when it wrote x and all methods m_r on t_r's call stack when it reads x there exists some group (W, R) in the specification such that $m_w \in W$ and $m_r \in R$.*

3.4 Modularity: Communication Modules

Specifications composed of the communication inlining and group primitives suffice for simple programs, but for larger programs it is natural to specify communication at multiple layers of abstraction. Consider the simple producers-consumers pipeline sketched in Figure 3. Producer threads (not shown) call `produce` to produce items and enqueue them in a bounded buffer, and consumer threads (also not shown) call `consume` to dequeue and process items for the next stage of the pipeline.

Communication in this program occurs at two levels of abstraction. At a low level, the bounded buffer methods `enqueue` and `dequeue` communicate through a shared buffer representation:

$$G_{Buffer} = (\{\texttt{enqueue}, \texttt{dequeue}\}, \{\texttt{enqueue}, \texttt{dequeue}\})$$

At a higher level, `produce` communicates to `consume` in the pipeline by enqueueing items in a bounded buffer from which `consume` later dequeues them:

$$G_{Pipe} = (\{\texttt{produce}\}, \{\texttt{consume}\})$$

However, since the bounded buffer's `size` field is both read and written when `produce` calls `enqueue` *and* when consume calls dequeue, then, for example, consume may communicate to produce, so we must settle on the following specification until we introduce encapsulation in Section 3.5:

$$G_{Pipe} = (\{\texttt{produce}, \texttt{consume}\}, \{\texttt{produce}, \texttt{consume}\})$$

With only inlining and groups, we are stuck with three unsatisfactory specifications:

1. Specify only the low-level bounded-buffer abstraction by inlining the pipeline methods;
2. Specify only the higher-level pipeline abstraction by inlining the bounded buffer methods; or
3. Specify that all four methods are writers and readers in a single group that lacks any notion of abstraction at all:

$$\begin{aligned} G_{Pipe} = (&\{\texttt{produce}, \texttt{consume}, \texttt{enqueue}, \texttt{dequeue}\},\\ &\{\texttt{produce}, \texttt{consume}, \texttt{enqueue}, \texttt{dequeue}\}) \end{aligned}$$

To specify communication at multiple levels of abstraction, we introduce communication modules.

The Communication Module Primitive. A *communication module* consists of a set of related methods and a set of groups whose members are drawn from these methods. The methods in a module interact with each other and perform communication described by the module's groups to implement a communication abstraction such as the bounded buffer. The pipeline program in Figure 3 has two modules, which are aligned by default with Java packages. Explicit annotations of arbitrary modules are also supported; in this case the default suffices. The module M_p contains the methods and groups for the item processing pipeline, and the module M_b contains the methods and groups for the bounded buffer:

$$\begin{aligned} M_p &= (\{\texttt{produce}, \texttt{consume}\}, \{G_{Pipe}\})\\ G_{Pipe} &= (\{\texttt{produce}, \texttt{consume}\}, \{\texttt{produce}, \texttt{consume}\})\\ M_b &= (\{\texttt{enqueue}, \texttt{dequeue}\}, \{G_{Buffer}\})\\ G_{Buffer} &= (\{\texttt{enqueue}, \texttt{dequeue}\}, \{\texttt{enqueue}, \texttt{dequeue}\}) \end{aligned}$$

Section 3.5 discusses the @InterfaceGroup annotation.

The levels of communication abstraction in the pipeline program are well aligned with the program representation, but we must map them clearly to the dynamic communication behavior of the program as well. Recall our description in Section 3.3 that when the result of a write operation is read by a read operation in another thread, all methods on the call stack at the write operation communicate to all methods on the call stack at the read operation. This definition assumes a single monolithic communication abstraction, so we now extend it to support modularity. To support layered abstractions, we divide communicating stacks into segments corresponding to each layer.

A *stack segment* is a maximal contiguous sequence of methods on a single call stack that all belong to the same module. As an example, Figure 4(a) shows the segmented call stacks at the time of a pair of write and read operations in the pipeline program. On the writer stack, the lower segment, consisting of enqueue, belongs to the module M_b, while the upper segment, consisting of produce, belongs to M_p. The reader stack has corresponding segments, containing dequeue and consume, respectively.

The fact that the segments in these two communicating stacks segments align by module is key. We say that two stacks S_w and S_r have *equivalent segmentations* if both stacks have n segments and for all $i \in 1 \ldots n$ the ith segment on S_w belongs to the same module as the ith segment on S_r. Together, the ith segments on a pair of communicating call stacks with equivalent segmentations form a layer of communication abstraction. When two communicating stacks do not have equivalent segmentations, either the communica-

```
package p;
@Group("Pipe")
class ItemProcessingPipeline {
  b.BoundedBuffer pipe = new b.BoundedBuffer();
  @Writer({"Pipe"})
  void produce() {
    ...;  pipe.enqueue(...);  ...
  }
  @Reader({"Pipe"})
  void consume() {
    ...;  ... = pipe.dequeue();  ...
  }
}
package b;
@InterfaceGroup("BufferClient")
@Group("Buffer")
public class BoundedBuffer {
  Item[] buffer = new Item[10];
  int size = 0;
  ...
  @Writer({"Buffer", "BufferClient"})
  @Reader({"Buffer"})
  public synchronized void enqueue(Item i) {
    while (size == buffer.length) wait();
    buffer[...] = i;
    size++;
    notifyAll();
  }
  @Writer({"Buffer"})
  @Reader({"Buffer", "BufferClient", })
  public synchronized Item dequeue() {
    while (size == 0) wait();
    size--;
    notifyAll();
    return buffer[...];
  }
}
```

Figure 3. A pipeline application that uses a simplified bounded buffer to communicate between stages.

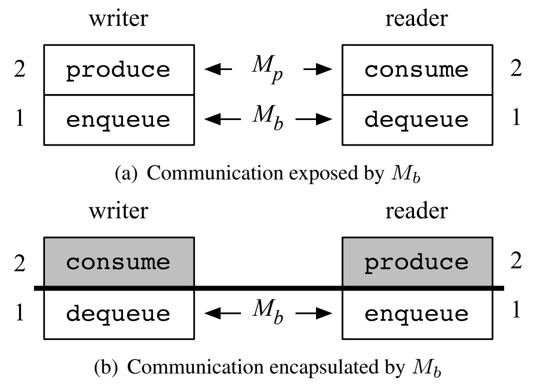

Figure 4. Segmented call stacks for a communication in the pipeline program from Figure 3. Each box is a segment. In this example all segments have exactly one method.

tion is in error, or the specification is in error and the misalignment of segments should be resolved by inlining more methods (recall that stacks contain no inlined methods).

Definition 2 (Valid Modular Communication)**.** *A memory write operation is allowed to communicate to a memory read operation if the stack at the write operation and the stack at the read operation have equivalent segmentations and for each segment on the writer stack, all methods in the segment are allowed to communicate to all methods in the corresponding segment on the reader stack, established by writer-reader group membership, as in Definition 1.*

Thus the pair of stacks in Figure 4(a) represents a valid communication, since `enqueue` is allowed to communicate to `dequeue` according to G_{Buffer} and `produce` is allowed to communicate to `consume` according to G_{Pipe}.

Communication modules express communication abstractions naturally without specifying extraneous communication across abstraction boundaries. Furthermore, modules allow for incremental and composable specifications. Any program may use the bounded buffer implementation without any additional specification, leaving all other methods inlined. The bounded buffer is still checked for valid communication without placing restrictions on communication performed in the rest of the program.

3.5 Encapsulation: Interface Groups

With inlining, groups, and communication modules, we are still unable to express the ideal specification for the pipeline abstraction, reflecting our intuition that `produce` communicates to `consume` and no other communication is possible:

$$G_{Pipe} = (\{\texttt{produce}\}, \{\texttt{consume}\})$$

The missing link is the encapsulation of communication from `dequeue` to `enqueue` (via the `size` field and the `this` lock) in the bounded buffer module. To encapsulate communication in modules, we introduce interface groups.

The Interface Group Primitive. *Interface groups* are a primitive for communication encapsulation in modules. We extend communication modules to include a set of interface groups in addition to their member methods and communication groups. Like communication groups, interface groups are composed of writer and reader methods within the module. While communication groups describe what communication is allowed among methods in the module, interface groups describe what communication is exposed to external callers of the module's methods.

Returning to the pipeline example, we declare an interface group for the bounded buffer:

```
@InterfaceGroup("BufferClient")
```

We annotate `enqueue` as a writer and `dequeue` as a reader in this group, meaning that communication from `enqueue` to `dequeue` will be exposed to their callers, but all other communication (e.g., from `dequeue` to `enqueue`) is encapsulated by the module and not exposed to callers. The result yields a clean interface for the bounded buffer and the intuitive specification for the pipeline:

$$\begin{aligned} M_p &= (\{\texttt{produce}, \texttt{consume}\}, \{G_{Pipe}\}, \emptyset) \\ G_{Pipe} &= (\{\texttt{produce}\}, \{\texttt{consume}\}) \\ M_b &= (\{\texttt{enqueue}, \texttt{dequeue}\}, \{G_{Buffer}\}, \{I_{BufferClient}\}) \\ G_{Buffer} &= (\{\texttt{enqueue}, \texttt{dequeue}\}, \{\texttt{enqueue}, \texttt{dequeue}\}) \\ I_{BufferClient} &= (\{\texttt{enqueue}\}, \{\texttt{dequeue}\}) \end{aligned}$$

At the pipeline level, we can now regard the bounded buffer just as we do memory. An `enqueue` operation and a `dequeue` operation may result in communication just as a write operation and read operation would. Alternatively, memory may now be regarded as simply one more lowest layer of communication abstraction with an interface dictating that the "write location method" (i.e., writes) communicates to the "read location method" (i.e., reads).

We extend Definition 2 to define when a communication violates a specification in the presence of encapsulation:

Definition 3 (Valid Communication)**.** *We say that communication is encapsulated by a corresponding pair of segments if the pair of segments are at the roots of their stacks or if no interface group contains the deepest (caller-most) method in the writer segment as a writer and the deepest method on the reader segment as a reader.*

Communication between two stacks S_w and S_r is allowed by a specification if there exist stack prefixes S'_w and S'_r composed of segments $1 \ldots k$ of S_w and S_r, respectively, such that communication is encapsulated by the kth pair of segments and communication from S'_w to S'_r is allowed by Definition 2.

When `dequeue` communicates to `enqueue`, the communication is encapsulated by module M_b. As shown in Figure 4(b), the communication is not exposed above M_b's layer on the two stacks; even though `consume` is not allowed to communicate to `produce`, this communication is valid. In this case, it happens that the entire stacks have equivalent segmentations, even above the encapsulation boundary. In general, however, this is not required.

In reality, some communication abstractions, such as those involving callbacks, do not map so clearly to layered abstractions. Callbacks may cause communication between a stack with direct control and a stack with inverted control that has an "extra" segment for the callback caller at its root. While not a perfect match, we find that inlining callback systems is a reasonable way to address this pattern. We discuss one example in Section 6.3.

4. Formal Semantics

In this section we present a formal semantics for simple multithreaded programs to gain a precise definition of when a program execution satisfies or violates a communication specification, and briefly discuss salient properties of communication-checked programs. We observe that shared-memory communication has a very restricted interaction with program semantics: only memory accesses and method entry and exit are relevant. As a result, we use a simplified view of the execution of multithreaded programs, in which each thread is reduced to a trace of operations on global and local state, eliding details that do not affect inter-thread communication.

Our formalization has three key parts. The first describes a simple operational semantics for the execution of multithreaded programs. The second part is an operational semantics for the execution of communication-checked multithreaded programs that effectively instruments the simple semantics with the necessary bookkeeping and checking. This instrumented semantics is defined in terms of the third component of the semantics: a separate judgment that captures the semantics of when a dynamic memory write operation is allowed to communicate to a dynamic memory read operation, based on the call stacks when the operations occurred.

4.1 Standard Multithreaded Semantics

A multithreaded program is a set of threads executing concurrently, each identified by a unique identifier t and accompanied by a thread state π, representing all thread-local storage and state information. The thread state store θ maps each thread's identifier to its state. The threads share a heap that maps each variable x to a value v and each lock l to the identifier of the thread that holds it, or $\bot$ if no thread does.

A program state $\sigma = (H, \theta)$ is comprised of a heap and a thread state store. Threads change the program state by performing operations a that update the heap and the thread state. These operations are reading from or writing to a shared variable x in the heap ($\mathsf{rd}(x, v)$ and $\mathsf{wr}(x, v)$), acquiring or releasing a lock l ($\mathsf{acq}(l)$ and $\mathsf{rel}(l)$), and entering or exiting a method m ($\mathsf{enter}(m)$ and $\mathsf{exit}(m)$).

Programs:

Thread ID	t	Thread State	π
Method ID	m	Variable	x
Lock	l	Value	v
Address	$p ::= x \mid l$	Holder	$ls ::= t \mid \bot$

Heap	$H ::= \cdot \mid H, x \mapsto v \mid H, l \mapsto ls$
Thread Map	$\theta ::= \cdot \mid \theta, t \mapsto \pi$
State	$\sigma ::= (H, \theta)$
Operation	$a ::= \mathsf{wr}(x, v) \mid \mathsf{rd}(x, v) \mid \mathsf{acq}(l) \mid \mathsf{rel}(l) \mid \mathsf{enter}(m) \mid \mathsf{exit}(m)$

Instrumentation:

Shadow Stack	$S ::= \cdot \mid S, m$
Instrumented Thread Map	$\Theta ::= \cdot \mid \Theta, t \mapsto (\pi, S)$
Last Writer Map	$\phi ::= \cdot \mid \phi, p \mapsto (t, S)$
Instrumented State	$\Sigma ::= (H, \phi, \Theta)$

Specifications and checking:

Method Set	$\mu, R, W ::= \{m_1, \ldots, m_n\}$
Stack Segment	$\hat{S} ::= \{m_1, \ldots, m_n\}$
Group	$G ::= (W, R)$
Group Set	$\gamma ::= \{G_1, \ldots, G_n\}$
Module	$M ::= (\mu, \gamma_C, \gamma_I)$
Specification	$\Gamma ::= \{M_1, \ldots, M_n\}$

Figure 5. Domains.

Operations a executed by thread t may update the heap as shown in the judgment $H; t; a \rightarrow H'$ in Figure 6. Reads and writes act as expected; lock acquire and release update the lock's heap entry to reflect its holder or its unheld status. Method entry and exit have no effect on the heap. We represent constraints on the possible steps a thread can take, such as program order, control flow, and data flow, by a relation *Program* over thread identifiers, initial thread states, operations, and the resulting thread states. Thread t, starting in state π, can execute operation a, ending in state π' when $\mathit{Program}(t, \pi, a, \pi')$ holds.

A program can step from state σ to σ' by nondeterministically selecting a thread to perform an operation that satisfies the *Program* relation and can execute on the current heap, as shown in the judgment $\mathit{Program} \vdash \sigma \rightarrow \sigma'$ in Figure 6.

4.2 Communication-Checked Semantics

To check communication in a program against a specification Γ, we instrument the standard multithreaded semantics

$$\boxed{H; t; a \to H'}$$

$$\textsc{write}\ \frac{}{H; t; \mathsf{wr}(x, v) \to (H, x \mapsto v)} \qquad \textsc{read}\ \frac{H(x) = v}{H; t; \mathsf{rd}(x, v) \to H} \qquad \textsc{acquire}\ \frac{H(l) = \bot}{H; t; \mathsf{acq}(l) \to (H, l \mapsto t)} \qquad \textsc{release}\ \frac{H(l) = t}{H; t; \mathsf{rel}(l) \to (H, l \mapsto \bot)}$$

$$\textsc{enter}\ \frac{}{H; t; \mathsf{enter}(m) \to H} \qquad \textsc{exit}\ \frac{}{H; t; \mathsf{exit}(m) \to H}$$

$$\boxed{Program \vdash \sigma \to \sigma'}$$

$$\textsc{step}\ \frac{Program(t, \theta(t), a, \pi') \qquad H; t; a \to H'}{Program \vdash (H, \theta) \to (H', (\theta, t \mapsto \pi'))}$$

Figure 6. Operational semantics for multithreaded programs.

$$\boxed{\Gamma \vdash \phi; t; S; a \Rightarrow \phi'; S'}$$

$$\textsc{ins enter}\ \frac{(\mu, \gamma_C, \gamma_I) \in \Gamma \qquad m \in \mu}{\Gamma \vdash \phi; t; S; \mathsf{enter}(m) \Rightarrow \phi; S, m} \qquad \textsc{ins exit}\ \frac{(\mu, \gamma_C, \gamma_I) \in \Gamma \qquad m \in \mu}{\Gamma \vdash \phi; t; S, m; \mathsf{exit}(m) \Rightarrow \phi; S} \qquad \textsc{ins inlined enter}\ \frac{\forall (\mu, \gamma_C, \gamma_I) \in \Gamma\ .\ m \notin \mu}{\Gamma \vdash \phi; t; S; \mathsf{enter}(m) \Rightarrow \phi; S}$$

$$\textsc{ins inlined exit}\ \frac{\forall (\mu, \gamma_C, \gamma_I) \in \Gamma\ .\ m \notin \mu}{\Gamma \vdash \phi; t; S; \mathsf{exit}(m) \Rightarrow \phi; S} \qquad \textsc{ins write}\ \frac{}{\Gamma \vdash \phi; t; S; \mathsf{wr}(x, v) \Rightarrow (\phi, x \mapsto (t, S)); S} \qquad \textsc{ins thread-local read}\ \frac{\phi(x) = (t, S')}{\Gamma \vdash \phi; t; S; \mathsf{rd}(x, v) \Rightarrow \phi; S}$$

$$\textsc{ins communicating read}\ \frac{t \neq t' \qquad \phi(x) = (t', S') \qquad \Gamma \vdash S' \rightsquigarrow S}{\Gamma \vdash \phi; t; S; \mathsf{rd}(x, v) \Rightarrow \phi; S} \qquad \textsc{ins release}\ \frac{}{\Gamma \vdash \phi; t; S; \mathsf{rel}(l) \Rightarrow (\phi, l \mapsto (t, S)); S} \qquad \textsc{ins thread-local acquire}\ \frac{\phi(l) = (t, S')}{\Gamma \vdash \phi; t; S; \mathsf{acq}(l) \Rightarrow \phi; S}$$

$$\textsc{ins communicating acquire}\ \frac{t \neq t' \qquad \phi(l) = (t', S') \qquad \Gamma \vdash S' \rightsquigarrow S}{\Gamma \vdash \phi; t; S; \mathsf{acq}(l) \Rightarrow \phi; S}$$

$$\boxed{Program; \Gamma \vdash \Sigma \Rightarrow \Sigma'}$$

$$\textsc{checked step}\ \frac{\Theta(t) = (\pi, S) \qquad Program(t, \pi, a, \pi') \qquad H; t; a \to H' \qquad \Gamma \vdash \phi; t; S; a \Rightarrow \phi'; S'}{Program; \Gamma \vdash (H, \phi, \Theta) \Rightarrow (H', \phi', (\Theta, t \mapsto (\pi', S')))}$$

Figure 7. Operational semantics for communication-checked multithreaded programs.

to maintain a *shadow stack* S for each thread and a global map ϕ recording the *last writer* of every variable (and the last releaser of every lock). A specification Γ is a set of modules $M = (\mu, \gamma_C, \gamma_I)$, where μ is the set of methods that belong to the module. Specifications and modules are discussed in more detail in Section 4.3. For now it suffices to understand that inlined methods do not belong to any module.

The judgment $\Gamma \vdash \phi; t; S; a \Rightarrow \phi'; S'$, in Figure 7, describes the effects of operations on shadow stacks and the last-writers map. A program can take a step according to this judgment if the step will not violate the specification.

A shadow stack S represents a thread's call stack, with inlined methods elided. When thread t enters the non-inlined method m, it pushes m onto its shadow stack, popping it off when it exits the method. Inlined methods are ignored. (See rules INS ENTER, INS EXIT, INS INLINED ENTER, and INS INLINED EXIT.) When a thread's current shadow stack is $\cdot, m_1, m_2$, its program counter is in m_2 (or an inlined transitive callee of m_2), where m_2 was called by m_1 (or an inlined transitive callee of m_1), and m_1 was the thread's entry point, or a transitive callee of an inlined entry point.

The last-writers map ϕ stores for each variable and lock the last thread to write or release it and that thread's shadow stack at the time of the operation. The entry in the last-writers map for a variable x or lock l is updated with the executing thread and its current stack on every write to x or release of l, as shown in rules INS WRITE and INS RELEASE.

$$\boxed{\mu; m \vdash S = S'; \hat{S}}$$

COLLECT
$$\frac{m \in \mu \qquad \mu; m' \vdash S = S'; \hat{S}}{\mu; m' \vdash S, m = S'; \hat{S} \cup \{m\}}$$

BORDER
$$\frac{m \in \mu \qquad S = S', m' \qquad m' \notin \mu}{\mu; m \vdash S, m = S; \{m\}}$$

END
$$\frac{m \in \mu}{\mu; m \vdash \cdot, m = \cdot; \{m\}}$$

$$\boxed{\gamma \vdash m_w \rightsquigarrow m_r}$$

CHECK METHODS
$$\frac{(W, R) \in \gamma \qquad m_w \in W \qquad m_r \in R}{\gamma \vdash m_w \rightsquigarrow m_r}$$

$$\boxed{\gamma_C \vdash \hat{S}_w \rightsquigarrow \hat{S}_r}$$

CHECK SEGMENTS
$$\frac{\forall m_w \in \hat{S}_w \,.\, \forall m_r \in \hat{S}_r \,.\, \gamma_C \vdash m_w \rightsquigarrow m_r}{\gamma_C \vdash \hat{S}_w \rightsquigarrow \hat{S}_r}$$

$$\boxed{\Gamma \vdash S_w \rightsquigarrow S_r}$$

EXPOSED LAYER
$$\frac{(\mu, \gamma_C, \gamma_I) \in \Gamma \qquad \mu; m_w \vdash S_w = S'_w; \hat{S}_w \qquad \mu; m_r \vdash S_r = S'_r; \hat{S}_r \qquad \gamma_C \vdash \hat{S}_w \rightsquigarrow \hat{S}_r \qquad \gamma_I \vdash m_w \rightsquigarrow m_r \qquad \Gamma \vdash S'_w \rightsquigarrow S'_r}{\Gamma \vdash S_w \rightsquigarrow S_r}$$

ENCAPSULATED LAYER
$$\frac{(\mu, \gamma_C, \gamma_I) \in \Gamma \qquad \mu; m_w \vdash S_w = S'_w; \hat{S}_w \qquad \mu; m_r \vdash S_r = S'_r; \hat{S}_r \qquad \gamma_C \vdash \hat{S}_w \rightsquigarrow \hat{S}_r \qquad \gamma_I \nvdash m_w \rightsquigarrow m_r}{\Gamma \vdash S_w \rightsquigarrow S_r}$$

EMPTY
$$\frac{}{\Gamma \vdash \cdot \rightsquigarrow \cdot}$$

Figure 8. Stack checking semantics.

Read and acquire operations are instrumented to check that any communication they complete is allowed by the specification Γ. Thread-local reads and acquires are always allowed. When t_r reads from variable x, if the last thread to write to x was t_r, then the read is allowed to proceed, as shown in rule INS THREAD-LOCAL READ. If the last thread t_w to write to x was not the same as the thread t_r executing the read operation, then the read is only allowed to proceed if the specification Γ allows communication from the stack S_w of t_w at the time it wrote to x to t_r's current stack, S_r, according to the judgment $\Gamma \vdash S_w \rightsquigarrow S_r$, in Figure 8. (Specifications and stack checking are described in detail in Section 4.3.) The same logic applies to lock acquires, with respect to the last release of the same lock.

An instrumented program state Σ consists of a heap H, a last-writers map ϕ, and an instrumented thread state store Θ, which maps each thread identifier to the associated thread state, instrumented with a shadow stack. An instrumented program is allowed to step from one instrumented state to another under the communication-checked semantics, as shown in the judgment $\mathit{Program}; \Gamma \vdash \Sigma \Rightarrow \Sigma'$, in Figure 7, if it can step under the simple semantics *and* the operation is allowed under the instrumented semantics by satisfying the judgment $\Gamma \vdash \phi; t; S; a \Rightarrow \phi'; S'$. Specifically, the rule CHECKED STEP ensures that read and acquire operations are only possible when the communication they complete is thread-local or allowed by the specification.

4.3 Specification Semantics

A communication specification Γ is a set of modules. Each module $M = (\mu, \gamma_C, \gamma_I)$ consists of a set of methods μ that does not overlap with that of any other module in the specification, a set of communication groups γ_C that describes what communication is allowed among methods in μ, and a communication interface γ_I that describes what communication among methods in μ is visible to callers outside the module. A communication group G is a pair (W, R) of sets of methods, denoting communication from every method in the writer set W to every method in the reader set R.

A *stack segment* $\hat{S}$ is the set of methods appearing in a maximal contiguous subsequence of a stack such that all of the methods in the subsequence belong to the same module. A stack segment represents a conceptual layer of communication abstraction in a communicating stack. The judgment $\mu; m \vdash S = S'; \hat{S}$, in Figure 8, states that a stack S is prefixed by a maximal non-empty sequence of methods comprising the segment $\hat{S}$, where $\hat{S} \subseteq \mu$, m is the method in $\hat{S}$ that is deepest in the stack prefix, and S' is the suffix of S starting with the shallowest method m' on S such that $m' \notin \mu$. If all methods on S are in μ, then S' is the empty stack. Though the rules implementing this judgment are somewhat subtle, with a recursive case (COLLECT), and two base cases (BORDER and END) for collecting stack segments, they work together to select one segment of contiguous methods belonging to one module from the callee end of the stack. Applied recursively, this judgment just segments a stack as described in Section 3.4.

Communication from writer stack S_w to reader stack S_r is checked recursively by the judgment $\Gamma \vdash S_w \rightsquigarrow S_r$ in Figure 8 by peeling stack segments $\hat{S}_w$ and $\hat{S}_r$ from the tips of S_w and S_r, respectively, where m_w and m_r are the deepest methods in the two segments, and S'_w and S'_r are the suffixes of the two stacks, respectively.

We check that $\hat{S}_w$ and $\hat{S}_r$ belong to the same module $M = (\mu, \gamma_C, \gamma_I)$ and that for all methods $m'_w \in \hat{S}_w$ and all methods $m'_r \in \hat{S}_r$, there exists some group in γ_C in which m'_w is a writer and m'_r is a reader. (Recall that stacks contain no inlined methods.) These checks are described by the judgments $\gamma_C \vdash \hat{S}_w \rightsquigarrow \hat{S}_r$ and $\gamma \vdash m'_w \rightsquigarrow m'_r$. If this segment matching fails, then communication from S_w to S_r is prohibited by the specification. Otherwise, if m_w and m_r, the segment boundary methods, are not writer and reader respectively in any group in the module's communication interface γ_I, then communication is hidden from S'_w and S'_r by this pair of segments, so no more checking is necessary; communication from S_w to S_r is allowed. Otherwise, if some group in the module's communication interface does describe the writer-reader pair (m_w, m_r), then communication is exposed by these segments and we must recursively check that the two stack suffixes S'_w and S'_r are allowed to communicate.

4.4 Properties of Communication-Checked Programs

Although the main purpose of our formalization is to define precisely when a dynamic memory operation satisfies or violates a communication specification, an ancillary benefit is that we can reason about important properties of communication-checked program executions. In this section, we sketch, but for brevity do not prove, meta-theorems on the equivalence of communication-checked and uninstrumented program executions (with a special equivalence case for empty specifications) and the soundness and precision of communication checking.

When describing program executions, we use $\rightarrow^*$ and $\Rightarrow^*$ to denote the reflexive and transitive closure of $\rightarrow$ and $\Rightarrow$, respectively. To compare uninstrumented and communication-checked executions, we must first define the equivalence of uninstrumented and instrumented program states. If $\forall t \,.\, \exists S \,.\, \Theta(t) = (\theta(t), S)$, then (H, θ) is equivalent to (H, ϕ, Θ) and we write $(H, \theta) \equiv (H, \phi, \Theta)$ or $(H, \phi, \Theta) \equiv (H, \theta)$. The initial program state σ_0 and the initial instrumented program state Σ_0 are equivalent. Both hold the empty heap and the same initial thread states. Σ_0 also holds the empty last-writers map, and stores the empty stack for each thread.

Equivalence of Semantics. All executions admitted by a communication-checked program are also admitted by the uninstrumented program; given a specification, all executions admitted by the uninstrumented program are either admitted by the communication-checked program too or contain communication that is invalid under the specification:

1. If $\mathit{Program}; \Gamma \vdash \Sigma_0 \Rightarrow^* \Sigma$ then $\exists \sigma \equiv \Sigma$ such that $\mathit{Program} \vdash \sigma_0 \rightarrow^* \sigma$.
2. Given Γ, if $\mathit{Program} \vdash \sigma_0 \rightarrow^* \sigma$ then either:
   (a) $\exists \Sigma \equiv \sigma$ such that $\mathit{Program}; \Gamma \vdash \Sigma_0 \Rightarrow^* \Sigma$ and all communication in $\mathit{Program} \vdash \sigma_0 \rightarrow^* \sigma$ is valid under Γ, or
   (b) $\nexists \Sigma \equiv \sigma$ such that $\mathit{Program}; \Gamma \vdash \Sigma_0 \Rightarrow^* \Sigma$ and $\mathit{Program} \vdash \sigma_0 \rightarrow^* \sigma$ contains communication that is invalid under Γ.

Equivalence of Semantics Under the Empty Specification. A special case is that, given a specification where all methods are inlined, a communication-checked program admits all the executions admitted by the uninstrumented program:

If $\mathit{Program} \vdash \sigma_0 \rightarrow^* \sigma$, then $\exists \Sigma \equiv \sigma$ such that $\mathit{Program}; \emptyset \vdash \Sigma_0 \Rightarrow^* \Sigma$.

Soundness. If an uninstrumented program execution performs communication that is invalid under a given specification, then the communication-checked version does not admit that execution, given the specification:

Given Γ, if $\mathit{Program} \vdash \sigma_0 \rightarrow^* \sigma$ and the uninstrumented program execution performs communication that is invalid under Γ then $\nexists \Sigma \equiv \sigma$ such that $\mathit{Program}; \Gamma \vdash \Sigma_0 \Rightarrow^* \Sigma$.

Precision. A communication-checked program admits all executions admitted by the uninstrumented version that do not perform invalid communication under its specification:

Given Γ, if $\mathit{Program} \vdash \sigma_0 \rightarrow^* \sigma$ and $\nexists \Sigma \equiv \sigma$ such that $\mathit{Program}; \Gamma \vdash \Sigma_0 \Rightarrow^* \Sigma$, then $\mathit{Program} \vdash \sigma_0 \rightarrow^* \sigma$ contains communication that is invalid under Γ.

5. Implementation

In this section we describe OSHAJAVA,[1] our prototype implementation of communication specifications for Java. OSHAJAVA specifications are expressed by Java annotations. At class load time, we use bytecode instrumentation to instrument each write operation with a communication state update and each read operation with a check to see if the method communication it causes obeys the specification. The instrumentation causes the program to throw a communication exception if its next step would violate its specification. Though deeper compiler or virtual machine integration might afford more optimization opportunities, our implementation offers the following useful properties:

- **Portability:** OSHAJAVA programs compile and run with any Java 1.6-compatible compiler and virtual machine.
- **Interoperability:** Every valid Java program is also a valid OSHAJAVA program (and vice versa) at both the source and bytecode levels. Every Java program can run unmodified and without recompilation under OSHAJAVA (modulo performance overhead) and every OSHAJAVA program can run unmodified and without recompilation under Java (without runtime specification checking).
- **Flexibility:** Programmers can annotate programs incrementally and mix unannotated and OSHAJAVA-annotated

[1] "OSHA" stands for Organized Sharing, the project's original working title.

components indiscriminately in programs running under OSHAJAVA or the standard unchecked Java runtime.

The remainder of this section describes the annotation system and the runtime system in more detail.

5.1 Specification Annotation System

By default, each Java package is a communication module. In practice, we observe that most modules align with packages or classes. However, programmers may define their own modules comprised of arbitrary sets of methods. In the presence of method overloading, group identifiers are simpler to express than method identifiers. The annotation processor, run as a plugin to the Java compiler, compiles the specification for each module to an efficient form for runtime use. The task of instrumenting the program with specification checks is deferred to runtime to avoid compiling both instrumented and uninstrumented versions of programs.

Subtyping and Dynamic Dispatch. Our specification language and dynamic checker work on methods that are actually called at run-time, so we do not need any special support for method overriding. As a program design matter, one could argue that an overriding method should perform no more communication external to callers than is specified by the overridden method—this is just an instance of behavioral subtyping [25]—but we do not require the specifications for the methods to obey this relation. As a practical matter, a `@Super` annotation to indicate, "the same specification as the method being overridden" would work fine, but we have not suffered from its absence in our experience.

5.2 Runtime System

At runtime, OSHAJAVA instruments each class as it is loaded by the JVM. Each field f of each object o is tracked by a communication state field, inserted by the instrumentor, that stores the last thread to write a value $o.f$ and the call stack under which the write was performed. When a thread reads $o.f$, the runtime first checks the communication state for $o.f$ to see if the last write was performed by the same thread or if communication is allowed from the last-writer call stack to the current call stack. Checking the latter property uses the natural algorithmic version of stack checking as described in Definition 3 and Section 4.3, incrementally checking that the two stacks have equivalent segmentations and that all communication in each corresponding segment is allowed by some group in the specification. Performing the full stack check on every read is prohibitively expensive; fortunately, most memory reads can be checked by simpler means.

Checking Optimizations. To avoid the high cost of a full stack walk for every communicating read, we employ the following series of progressively more expensive checks. Each stack has an integer ID and a bit set of the IDs of other stacks that are allowed to communicate *to* this stack. This set is populated lazily as the program runs. In addition to the specification, the runtime maintains a global hash table-based memo table of pairs of stacks that have previously been checked. The checks proceed as follows:

1. If the last write was done by the same thread, the read does not communicate, and is trivially valid.
2. If the writer stack's ID is a member of the reader stack's bit set of valid writers, then the communication is valid.
3. If the pair of writer and reader stacks is in the global memo table of valid communicating stacks, then the communication is valid. If the number of checks of this pair of stacks that have been satisfied by the global memo table reaches a certain threshold (currently, 8), then the writer stack is given the next available non-zero ID, and this ID is added to the reader stack's bit set of valid writers. ID and bit set updates are synchronized with respect to each other on a given stack, but not with respect to bit set membership tests. At worst a membership test that should succeed races with an update and fails, reverting to a more expensive check.
4. If the writer stack has never communicated to the reader stack before, a full stack walk is performed. If this check fails, a communication exception is thrown, otherwise the pair is added to the global memo table.

Section 6.2 discusses the frequency with which each of these stages is used in practice. In summary, nearly all read operations are thread-local or validated by the bit sets. In typical programs, pairs of call stacks communicate repeatedly, so memoization quickly pays off. Allocating IDs lazily keeps the bit sets small. In practice, at most 41 inlined call stacks received IDs in any single execution. Our implementation uses bit sets that can grow to arbitrary size, but for all of the executions we have observed, a single 64-bit `long` would suffice. The thread-local check and bit set test are inlined into the body of the method performing the checked read.

Other Optimizations. A program may generate very large numbers of call stacks, visit the same call stack repeatedly, and visit many call stacks with differing tips but identical tails. Omitting inlined methods from shadow stacks reduces the size and number of shadow stacks we need to store and a hash-consing representation limits their duplication.

Each field and array element is tracked by a communication state storing the last writer to that field or to any element in that array. A runtime option enables array-level tracking for array accesses, with one communication state per array, trading precision for memory overhead. Object-level communication tracking is not yet implemented, mainly because the memory overheads we have observed with field-level tracking are reasonable. Tracking at the object and array granularities is sound and precise if, when thread t_r reads an element, all elements in the array are guaranteed to have been written last by the same thread t_w. This property is neither uncommon nor pervasive; where it holds, array-level tracking can save memory.

Atomicity of Checks and Accesses. Our tool relaxes the soundness requirement that the checks it inserts be atomic with the memory accesses they check. As a result, it is only fully sound and precise on race-free programs. However, the possibility for unsound communication checking behavior (based on out-of-date communication state) is limited to those data that were targets of races. Our experiences with various dynamic race detectors suggest that, in practice, forgoing strictly atomic check-access sequences yields substantial performance benefits, while unsound behavior occurs rarely if at all, an acceptable trade-off for a debugging tool. Ideally, synchronization is an orthogonal and separately checked concern.

6. Evaluation

The goals of our evaluation are to characterize our annotation language, understand the performance and memory overheads of the OshaJava checking tool, and discuss case studies. We used the multithreaded benchmarks from the Java Grande suite [37] and selected programs from version 9.12 of the DaCapo benchmark suite [5].

Table 2 shows the applications we annotated for our evaluation. The Java Grande benchmarks are relatively small (at most 1.2K lines of code), but exercise a variety of communication structures. From the DaCapo benchmark suite, which contains larger-scale parallel and concurrent programs, we examine a subset of the applications that exhibit significant communication in contrasting patterns. These applications were selected because they are representative of different patterns of communication. Avrora exhibits frequent and complex communication that often crosses module boundaries due to callback patterns. Batik is embarrassingly parallel; the only communication occurs at the top level in the test harness. Xalan communicates frequently but in a more modular way than Avrora.

While there are many possible valid specifications for a given program (even the empty specification—all methods inlined—suffices), we have attempted to annotate as thoroughly as possible. In particular, we have never inlined methods where meaningful communication seems to occur. However, programmers more familiar with the applications may construct specifications differently. For the small Java Grande benchmarks as well as Batik, we are very confident that we have annotated all meaningful communication; for the more complex Avrora and Xalan benchmarks, while there is greater chance that we have missed some meaningful communication, we believe we have covered the entire program, especially since unspecified communication would lead to exceptions.

We now present evaluations of: (1) specification size, or how many annotations were inserted; (2) specification precision, measuring how much of the specified communication was actually exercised; (3) runtime overheads of the checking tool in both time and space; and (4) a case study of Xalan and Avrora, the most complex applications we annotated.

6.1 Specification Size and Precision

Size of Annotations. Table 2 lists the number of annotations used in our communication specifications for each benchmark. The annotation count includes all of the annotations described in Section 3: group declarations, group memberships (multiple-group `@Reader` and `@Writer` declarations are considered multiple annotations), and explicit module memberships. The Java Grande benchmarks require between 0.5 and 1.5 annotations per method. These applications are very small and therefore an artificially large portion of their methods are involved in communication. In contrast, the DaCapo applications have a much lower frequency of annotations: one for every 50 to 100 methods.

Due to their size, our annotations of the Java Grande benchmarks consist of just one to three communication modules. In the DaCapo suite, Avrora has 7 non-empty modules (those with at least one non-inlined method) with an average of 12 non-inlined methods per module. Xalan has 6 modules with 7 non-inlined methods per module.

As expected, the majority of the methods in the benchmarks we examined could be inlined (and thus unannotated). In the Java Grande benchmarks, about 85% of the methods are inlined. Both Avrora and Xalan from DaCapo have greater than 99% of their methods inlined. Because our annotation system inlines methods by default, specifications can be created by identifying the small set of methods that must be annotated.

Dynamic Communication Characteristics. Our measure of specification precision is the proportion of the static specification dynamically exercised during a given execution of the program. Our annotation system allows a trade-off between conciseness and precision—by adding more annotations, the programmer can more tightly constrain the communication behavior of the program.

A good annotation system would provide high precision using a small number of annotations. However, complete precision may not be attainable or even desirable for all applications. Some looseness in specifications allows for variation in communication patterns across inputs. Loose specifications may also allow valid communication that does not currently occur but may start to occur as the program changes. Specifically, it may be helpful to allow communication between methods that share data and could, but never do, run on different threads. Such a specification corresponds well with intuition regarding the program's behavior and makes the specification robust to future changes in the program that run the methods on separate threads.

Figure 9 shows the proportion of methods and communicating method pairs declared in the specification that actually communicated at runtime. This is a direct measure

Name	Description	Lines of Code	Methods	Annotated Methods	Groups	Non-Empty Modules	Total Annotations
Crypt	IDEA encryption	300	17	5	4	1	16
LUFact	LU factorization	500	29	6	4	2	15
MolDyn	N-body simulation	500	27	16	6	2	39
MonteCarlo	Financial simulation	1200	172	11	3	1	19
RayTracer	3D ray-tracing	700	77	15	6	3	37
SOR	Linear system solver	200	13	5	3	1	14
Series	Fourier transform	200	15	6	2	1	10
SparseMatmult	Matrix multiplication	200	12	4	2	1	9
Avrora	Sensor network micro-controller simulator	70,000	9775	85	17	7	175
Batik	SVG image renderer	190,000	15547	8	2	2	16
Xalan	XSLT/XPath interpreter	180,000	7854	42	7	6	90

Table 2. Summary of benchmarks and their annotations. "Total Annotations" counts all the annotations described in Section 3. Non-empty modules have at least one non-inlined method. Lines of code were counted by David A. Wheeler's SLOCCount.

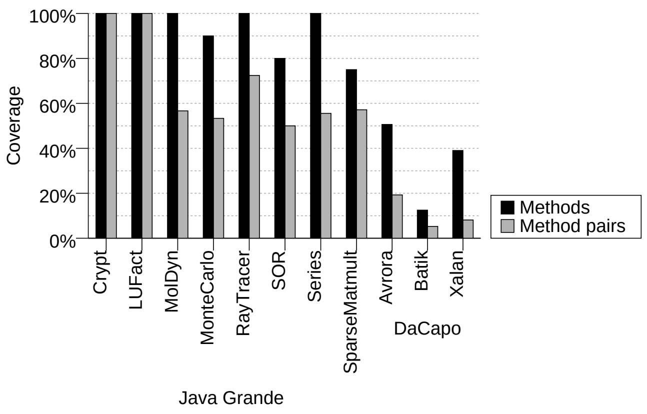

Figure 9. Proportion of the specification exercised during a single run of each benchmark on 8 threads. The first bar indicates the percentage of non-inlined methods that actually communicated with at least one other method. The second indicates the percentage of pairs of methods allowed to communicate that actually communicated.

of precision. As expected, the simpler applications (Java Grande) had a much higher proportion (≈80% vs. ≈30% on average) of communicating methods than the larger applications (DaCapo). The same applies to communicating pairs of methods (≈60% vs. ≈10% on average). This is largely due to the effect described earlier: the specifications conservatively allow communication between pairs of methods that would communicate if they ever ran on different threads. In Batik, only 1 method pair communicates out of 19 pairs allowed to communicate; all of the unexercised method pairs fall into the above category of methods that would communicate if they did not always run on the same thread. Recall that the annotations measured are first impressions by programmers unfamiliar with the applications details: further study could likely improve precision.

In order to make specification feasible, the number of methods in each stack segment (see Section 3.4) should be small: the programmer must allow every pair of reader and writer methods in a pair of stack segments to communicate, and the number of method pairs grows quickly with the size of module segments. A programmer using OSHAJAVA can keep these all-to-all checks small by dividing unrelated groups of methods into communication modules and by inlining most methods. For the Java Grande benchmarks, the average number of methods per stack segment is between 1 and 1.5. For two of the benchmarks (Crypt and LUFact), every segment had exactly 1 method. In DaCapo, Avrora has 1.8 methods per stack segment; Xalan's average is close to 1 while Batik's is exactly 1. Communication modules and inlining effectively keep OSHAJAVA's all-to-all checks small.

6.2 Performance

We ran performance and profiling experiments for the OSHAJAVA runtime on an 8-core 2.8GHz Intel Xeon E5462 machine with 10GB of memory, running Ubuntu GNU/Linux 8.10 and the HotSpot 64-bit client VM 1.6.0 with maximum heap size set to 8GB. We ran each benchmark 10 times in each configuration, measuring execution time and memory usage and preceding each set of 10 by a warmup run. For the Java Grande benchmarks, we used the largest available inputs; for DaCapo we used the default inputs.

Execution Time. Figure 10 shows the average execution time of benchmarks run on OSHAJAVA, configured to use 1, 2, 4, or 8 threads with element- and array-level communication tracking, normalized to the average execution time on Java *with the same number of threads*. The DaCapo benchmarks were run with default options (yielding 7 threads for Avrora, 8 for Batik, and 9 for Xalan). Single-threaded executions are included to demonstrate the baseline overheads introduced by OSHAJAVA. Overall, the performance is quite reasonable for a debugging tool. Most benchmarks experi-

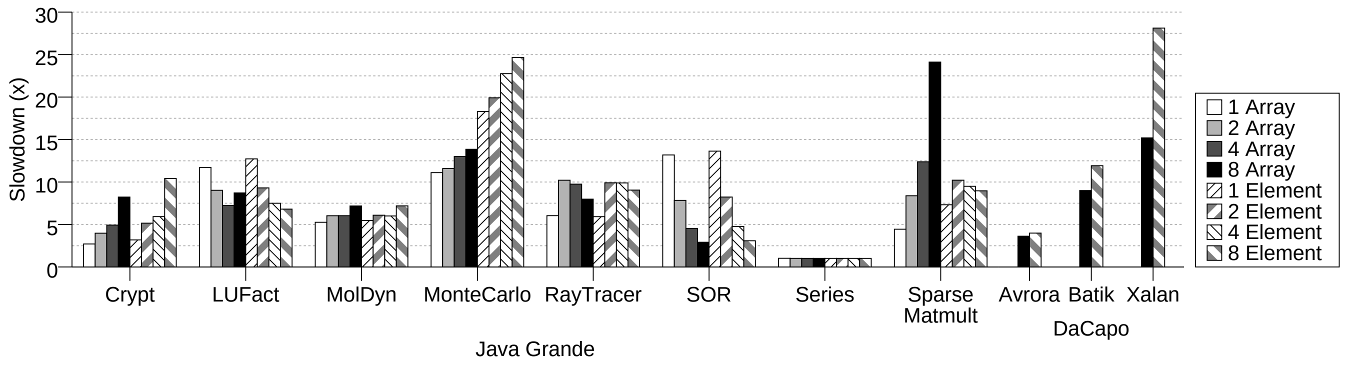

Figure 10. Average instrumented execution time of benchmarks with element- and array-level communication tracking for 1, 2, 4, and 8 threads, normalized to average uninstrumented execution time with the same number of threads.

ence 5-15x slowdown, trending towards the lower half of this interval. In general, performance with element-level tracking is on par with array-level tracking.

There are four major exceptions to these trends: Series, which has overheads between 0.1% and 2%, spends nearly all of its time inside a tight loop performing floating point operations on local variables, so slow field or array accesses are a non-issue. SparseMatmult scales very poorly with array-level communication tracking, running 24.1 times as long as under Java with 8 threads. However, it scales nicely with element-level tracking, suggesting that contention on shared array states is a bottleneck. Indeed, sparse matrix multiplication is known to be highly cache-sensitive. When many threads concurrently access different elements of an array under array-level tracking, caches contend heavily on the array's state, causing increased cache-coherence traffic, and likely affecting the scaling properties of SparseMatmult. MonteCarlo and Xalan run 1.5 to 2 times as slow with element-level tracking as with array-level tracking. This slowdown is due in part to garbage collection burden: MonteCarlo spends almost 4—and Xalan roughly 7–times longer garbage collecting with element-level tracking than with array-level tracking.

Memory Overhead. Figure 11 shows the average peak memory usage of benchmarks run on OSHAJAVA with array-level and array element-level communication tracking, normalized to the average peak memory usage on Java.[2] Memory overhead is fairly consistent across all numbers of threads; these data represent averages over runs with 8 threads. Array-level tracking overheads are quite low for all of the Java Grande benchmarks except MolDyn and MonteCarlo. Since many of these programs use arrays heavily, the overhead of element-level tracking is larger. Crypt's high memory overhead of 34.3x (roughly 5.4GB total) with element-level tracking results from its allocation of three 50,000,000-element byte arrays for the large input we tested.

[2] We believe RayTracer has a lower memory footprint running under OSHAJAVA because garbage collection is triggered more frequently.

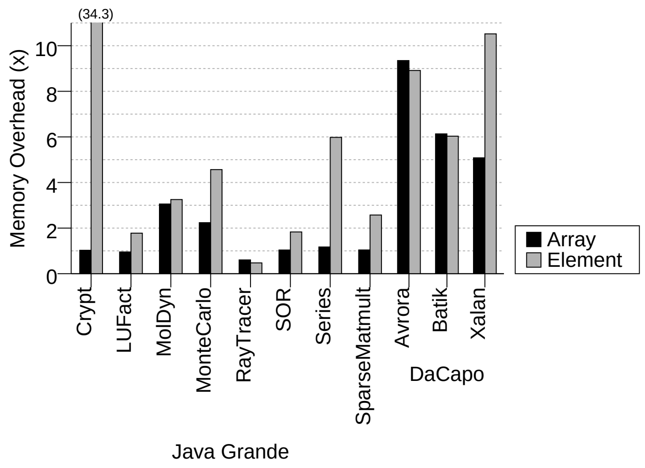

Figure 11. Average peak memory usage of instrumented executions with array- and element-level communication tracking, normalized to average peak memory usage of uninstrumented executions.

Effectiveness of Optimizations. We also profiled communication and checking in each of the benchmarks. For simplicity, we present this data aggregated across all thread configurations for each benchmark. Figure 12 shows the distribution of successful communication checks over the stage in the checking algorithm at which they succeeded. Communication checking slow paths are elided from this plot because they are not even visible when included. In fact, fewer than 1 in 10,000,000 reads were not thread-local or validated by the bit set memo table; less than two-thirds of these required stack walks. Across all benchmarks, the largest number of stack walks in a single execution was 697, while the number of communications ranged from a few thousand to 6 billion, with most executions performing tens of millions to hundreds of millions of communications. It is clear that aggressive memoization of valid pairs of communicating stacks makes dynamic communication checking tractable. Indeed, checking communication between deep stacks is no more costly than checking communication between empty stacks

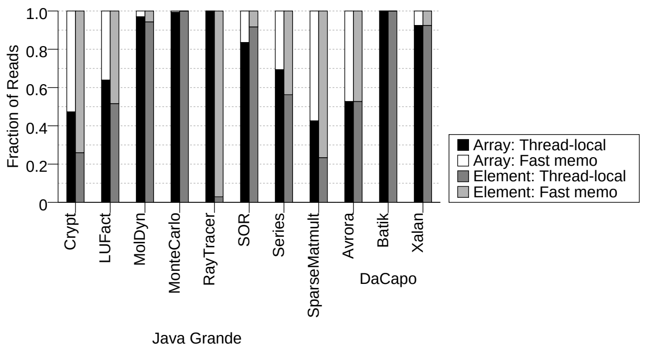

Figure 12. Fraction of all reads validated at each checking stage, for 8 threads with array- and element-level tracking.

since stack walks are performed exceedingly rarely. In the programs we have considered, performance is not affected by the depth or precision of a specification.

Figure 12 also illustrates soundness and precision issues with array-level communication tracking (as discussed in Section 5.2). In most of the Java Grande multithreaded benchmarks, array-level is imprecise. Communication structure in these programs is quite stable, so a difference in the rate of apparent communication under array- vs. element-level is a rough indicator of how much false communication or false non-communication occurs with array-level tracking. SOR observes more communications with array-level tracking, suggesting false communication. RayTracer falls at the opposite extreme: in reality, 97% of its reads communicate, but only 1% appear to communicate with array-level tracking. The explanation is simple: Under array-level tracking, when a thread t writes an element, it updates the array's last writer to t. If no other threads write elements of the array before t's next element read, it appears thread-local, regardless of which thread last wrote that particular element. This pattern is common in RayTracer.

Thus if memory is not a concern, there is no reason to sacrifice soundness and precision for array-level tracking. As observed previously, element-level tracking has insignificant performance impact in most of our benchmarks. The two exceptions to this trend, MonteCarlo and Xalan, exhibit little or no false communication or non-communication with array-level tracking.

6.3 Case Studies

We now discuss some of our experience annotating Xalan and Avrora, the two most complex applications we considered. We discovered communication properties by introducing restrictive specifications and successively relaxing them based on observed violations. We also used a simple tool that enumerates pairs of communicating methods. While programmers may approach the process differently when annotating programs incrementally as they are constructed, we believe our experience is representative of the process of annotating existing programs.

```
@Group("ListLinks")
class TransactionalList {
  Link head;
  Link tail;
  @Reader({"ListLinks"})
  @Writer({"ListLinks"})
  void add(Object value) {
    ...
    tail.next = new Link(value);
    tail = tail.next;
  }
}
class EventList extends TransactionalList {
  @Reader({"ListLinks"})
  void fireAll() {
    for (Link pos = head; pos != null;
         pos = pos.next)
      ((Event)pos.object).fire();
  }
}
```

Figure 13. A class in the Avrora benchmark from DaCapo that causes communication across callbacks. The desired annotations shown would prohibit this communication.

Simple Specifications. One advantage of our communication-centric approach to specification is its ability to distinguish which methods can write shared data. In data-centric systems such as SharC [3], shared data can be declared read-only to prevent modification after initialization, but the programmer cannot selectively allow read–write access to certain methods. The benchmarks we examined contained several methods that needed `@Reader` but not `@Writer` annotations, suggesting that this distinction is useful in specifying real communication patterns.

The large Java programs we examined often exhibited communication of object references that does not intuitively correspond to communication of data. For instance, many threads in Avrora use a global screen writer object. With strictly communication-centric annotations, every method that uses the screen writer must be allowed to read from the method that creates it. While this annotation strategy precisely describes the communication pattern, a system combining communication- and data-centric specification styles might allow the annotation to be more succinct by simply declaring the screen writer object read-only.

Modules and Abstraction. We found that, in the common case, communicating pairs of stacks have equivalent segmentations, as described in Section 3.4. Mismatched module sequences are in our experience confined to callback situations, in which a module appears once on one stack but twice on the other (i.e., two methods from one module are separated by at least one method from a different module). Avrora serves as a case study for this communication pattern.

In Figure 13, an `EventList` class extends a linked list class to invoke methods on simulation event objects at a later time. It would be desirable to place the list class in a separate module to avoid exposing incidental communication through its private fields. However, using this strategy, because the events fired by the `EventList` read data written outside of events, communication occurs from a stack without methods from the list module to one with such methods. This situation violates the assumption in Section 3.4 that communicating stack pairs have equivalent segmentations. This assumption prevents us from isolating `EventList` in a communication module, but we can still create a correct specification by inlining the list methods. This strategy yields a less precise specification but permits communication across callbacks.

In contrast to Avrora, Xalan exhibits modular, isolated communication that rarely crosses module boundaries. For example, Xalan includes a class called `FastStringBuffer` that matches well with OSHAJAVA's distinction between communication and interface groups. The buffer's `append` method must be allowed to communicate with itself because it both reads and writes private bookkeeping data. However, semantic communication occurs only from `append` to output methods like `toString`; clients of the class that only append to the buffer should not also be allowed to read it. We were able to succinctly specify this constraint by using distinct communication and interface specifications for the buffer module. Because of the modular nature of communication in Xalan, our specification for the benchmark does not need the inlining strategy described above for Avrora.

7. Future Work

While our approach to communication checking is entirely dynamic, an obvious area for future work is developing a sound, conservative static analysis for checking specifications. Our primary goal has been to develop a checkable and useful specification language that captures important safety properties of shared-memory programs in ways that match the program structure. We did not allow checking technology to unduly affect (e.g., reduce the expressiveness of) the specifications. Indeed, the need to check entire call stacks was a challenge our dynamic analysis had to overcome. Static analysis with reasonable precision will also face significant challenges, notably alias analysis for thread-shared data. We believe our program annotations and checking tool are valuable even without a static-analysis counterpart.

Currently, use of callbacks presents an inconvenience for our annotation system, requiring that methods be inlined when they should intuitively be encapsulated in a communication module. Extensions to our specification language may allow more precise specification in the case of callbacks and other situations when communicating stacks do not have equivalent segmentations. In addition, our language currently has no notion of logical threads—code units that should be considered "communicating" even when run in the same Java thread. As a result, our system does not check communication between distinct tasks that are multiplexed onto the same Java thread (e.g., in a worker thread pool). An additional language construct could address this issue.

Our specifications express properties distinct from those addressed by other approaches such as data-centric sharing specifications. Combining these approaches may offer opportunities for more precise specifications, including constraints that a given pair of methods may only communicate through a given shared field.

Communication specifications could be used for optimization of shared-memory programs. An operating system could use communication specifications to schedule communicating pieces of code to nearby processing units. Or, a compiler could better decide what optimizations to apply in threaded code if it knew the communication pattern.

Significant avenues exist for optimizing our runtime checker. Static escape analysis and other data flow analyses could likely identify several opportunities for sound instrumentation elision, improving performance significantly. A JIT compiler that performs thread-escape analysis will provide some of these benefits, but more static analysis could further reduce the overhead of instrumentation.

8. Related Work

Much of the work on checking properties of multithreaded software has focused on race detection and atomicity checking. Detecting data races is not a program-specific issue, so general tools requiring no annotations are successful, using either static [8, 12, 24, 29, 30, 33, 42, 43] or dynamic [9, 17, 36, 38, 41, 48, 49] analysis. That said, type systems and related annotation systems can make checking simpler and more modular [1, 6, 7, 13–15, 23]. Similarly, atomicity is a general property amenable to static [2, 18, 19, 44] and dynamic [16, 21, 32] approaches.

Our communication-centric approach complements this work in two ways: (1) it specifies how shared-memory communication occurs rather than data or isolation properties and (2) it focuses on program-specific properties with multiple levels of abstraction rather than generic properties. A generic property like "data-race free" or "atomic" in our communication-centric view would be "does no inter-thread communication," which we can easily specify. Such a specification—and finding violations of it—could well prove useful, but we have focused instead on specifications for methods that do communicate and checking that the specified communication encompasses all actual communication. Also note that "does no communication" is incomparable to both "data-race free" (write/write races do not communicate and communicating methods may or may not be racy) and atomic (an atomic method may not communicate and a communicating method may not be atomic).

In the sense of capturing simple program-specific multi-threading properties, some recent work shares many of our goals despite being data-centric. For example, SharC [3] lets programmers specify data-sharing rules at an object granularity: for example, *read-only* and *read-write*. Shoal [4] builds on SharC and lets programmers assign rules to entire data-structures, as opposed to individual objects. More recently, Martin et al. [27] present an annotation technique for C/C++ programs that lets programmers declare data-ownership properties (which threads own which data). As in our work, they use a dynamic tool to check specifications.

Approaches to verifying more sophisticated properties of multithreaded programs include abstraction and model-checking (e.g., [10, 46]) and modular theorem-prover techniques using assume-guarantee reasoning (e.g., [11, 20]). Richer notions of program verification naturally require more sophisticated annotations than our specifications; our goal is to focus on properties, namely communication, unique to multithreading, rather than more general program verification. Our work is complementary to model-checking techniques based on exhaustive concurrent program testing [22, 28, 40]: one could use exhaustive testing to determine (non)conformance to our specifications. Thread coloring [39] uses statically checked annotations to specify which threads are allowed to execute what code based on the roles they fulfill. The code communication properties that are specified and checked in our system are complementary to those verified by this approach.

In a rough sense, static pointer analysis for multithreaded programs [31, 34, 35] is related: If no memory accesses in methods m_1 and m_2 or any methods they call have overlapping points-to sets, then m_1 and m_2 definitely do not communicate via shared memory. However, points-to sets do not distinguish intra-thread and inter-thread communication, which is central to our work. Pointer analysis is also not naturally suited to working for multiple levels of abstraction and encapsulating communication within modules.

Recent work on code-based inter-thread communication invariants has employed hardware to record and analyze instruction-level inter-thread communication patterns in program executions for debugging [26] or inference of likely intended communication invariants to enforce in subsequent program executions [47]. Another approach records function-level communication for program understanding and characterization [45]. Our system is pure software and employs explicit specifications to define precisely what communication is allowed at the method level.

9. Conclusions

We have developed new communication-centric, simple, partial specifications for shared-memory multithreaded programs. The key idea is to specify which methods communicate with each other across threads. Essential to our technique is a treatment of (transitive) callees that is modular and allows specifications at multiple levels of abstraction. We have implemented a dynamic-analysis tool to check our specifications and shown that it is effective at checking non-trivial specifications for executions of benchmark programs.

Acknowledgments

This work was supported in part by an Anne Dinning - Michael Wolf Endowed Regental Fellowship, an ARCS Foundation Fellowship, a Hacherl Endowed Graduate Fellowship, and a Microsoft Faculty Fellowship. We thank Tom Bergan and Marius Nita for thoughts on modularity, and the SAMPA and WASP groups at the University of Washington for useful discussions. We also thank Steve Freund for helpful discussions on dynamic analysis techniques for concurrent programs.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for Safe Locking: Static Race Detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2), 2006.

[2] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized Run-time Race Detection and Atomicity Checking Using Partial Discovered Types. In *IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[3] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking Data Sharing Strategies for Multithreaded C. In *ACM Conference on Programming Language Design and Implementation*, 2008.

[4] Z. Anderson, D. Gay, and M. Naik. Lightweight Annotations for Controlling Sharing in Concurrent Data Structures. In *ACM Conference on Programming Language Design and Implementation*, 2009.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[6] C. Boyapati and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[8] G.-I. Cheng, M. Feng, C. Leiserson, K. Randall, and A. Stark. Detecting Data Races in Cilk Programs that Use Locks. In *ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[9] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *ACM Conference on Programming Language Design and Implementation*, 2002.

[10] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported Program Abstraction for Finite-state Verification. In *ACM/IEEE International Conference on Software Engineering*, 2001.

[11] T. Elmas, S. Qadeer, and S. Tasiran. A Calculus of Atomic Actions. In *ACM Symposium on Principles of Programming Languages*, 2009.

[12] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *ACM Symposium on Operating Systems Principles*, 2003.

[13] C. Flanagan and M. Abadi. Object Types Against Races. In *International Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[14] C. Flanagan and M. Abadi. Types for Safe Locking. In *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[15] C. Flanagan and S. N. Freund. Type-based Race Detection for Java. In *ACM Conference on Programming Language Design and Implementation*, 2000.

[16] C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *ACM Symposium on Principles of Programming Languages*, 2004.

[17] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *ACM Conference on Programming Language Design and Implementation*, 2009.

[18] C. Flanagan and S. Qadeer. A Type And Effect System For Atomicity. In *ACM Conference on Programming Language Design and Implementation*, 2003.

[19] C. Flanagan and S. Qadeer. Types for Atomicity. In *ACM Workshop on Types in Language Design and Implementation*, 2003.

[20] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular Verification of Multithreaded Programs. *Theoretical Computer Science*, 338(1–3), 2005.

[21] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound And Complete Dynamic Atomicity Checker for Multithreaded Programs. In *ACM Conference on Programming Language Design and Implementation*, 2008.

[22] P. Godefroid. Model Checking for Programming Languages Using Verisoft. In *ACM Symposium on Principles of Programming Languages*, 1997.

[23] D. Grossman. Type-Safe Multithreading in Cyclone. In *ACM Workshop on Types in Language Design and Implementation*, 2003.

[24] T. A. Henzinger, R. Jhala, and R. Majumdar. Race Checking by Context Inference. In *ACM Conference on Programming Language Design and Implementation*, 2004.

[25] B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[26] B. Lucia and L. Ceze. Finding Concurrency Bugs with Context-Aware Communication Graphs. In *ACM/IEEE International Symposium on Computer Architecture*, 2009.

[27] J.-P. Martin, M. Hicks, M. Costa, P. Akritidis, and M. Castro. Dynamically Checking Ownership Policies in Concurrent C/C++ Programs. In *ACM Symposium on Principles of Programming Languages*, 2010.

[28] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[29] M. Naik and A. Aiken. Conditional Must Not Aliasing for Static Race Detection. In *ACM Symposium on Principles of Programming Languages*, 2007.

[30] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *ACM Conference on Programming Language Design and Implementation*, 2006.

[31] M. G. Nanda and S. Ramesh. Pointer Analysis of Multithreaded Java Programs. In *ACM Symposium on Applied Computing*, 2003.

[32] C.-S. Park and K. Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *ACM International Symposium on the Foundations of Software Engineering*, 2008.

[33] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *ACM Conference on Programming Language Design and Implementation*, 2006.

[34] R. Rugina and M. C. Rinard. Pointer Analysis for Structured Parallel Programs. *ACM Transactions on Programming Languages and Systems*, 25(1), 2003.

[35] A. Salcianu and M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2001.

[36] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[37] L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *ACM/IEEE International Conference for High Performance Computing and Networking*, 2001.

[38] N. Sterling. A Static Data Race Analysis Tool. In *USENIX Winter Technical Conference*, 1993.

[39] D. F. Sutherland and W. L. Scherlis. Composable Thread Coloring. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2010.

[40] W. Visser, G. P. B. Klaus Havelund, and S. Park. Model Checking Programs. In *IEEE/ACM International Conference on Automated Software Engineering*, 2000.

[41] C. von Praun and T. Gross. Object Race Detection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[42] C. von Praun and T. R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *ACM Conference on Programming Language Design and Implementation*, 2003.

[43] J. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *ACM International Symposium on the Foundations of Software Engineering*, 2007.

[44] L. Wang and S. D. Stoller. Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2006.

[45] B. P. Wood, J. Devietti, L. Ceze, and D. Grossman. Code-Centric Communication Graphs for Shared-Memory Multithreaded Programs. Technical Report UW-CSE-09-05-02, University of Washington, 2009.

[46] E. Yahav. Verifying Safety Properties of Concurrent Java Programs Using 3-value Logic. In *ACM Symposium on Principles of Programming Languages*, 2001.

[47] J. Yu and S. Narayanasamy. A Case for an Interleaving Constrained Shared-Memory Multi-Processor. In *ACM/IEEE International Symposium on Computer Architecture*, 2009.

[48] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *ACM Symposium on Operating Systems Principles*, 2005.

[49] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *International Symposium on High-Performance Computer Architecture*, 2007.