Modern Concurrency Abstractions for C♯

NICK BENTON, LUCA CARDELLI and CÉDRIC FOURNET
Microsoft Research

Polyphonic C♯ is an extension of the C♯ language with new asynchronous concurrency constructs, based on the join calculus. We describe the design and implementation of the language and give examples of its use in addressing a range of concurrent programming problems.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed and parallel languages*; *Object-oriented languages*; *C♯*; D.3.4 [**Programming Languages**]: Processors—*Compilers*; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms: Design, Languages

Additional Key Words and Phrases: Asynchrony, chords, events, join calculus, messages, polyphonic C♯, synchronization, threads

1. INTRODUCTION

1.1 Languages and Concurrency

Concurrency is an important factor in the behaviour and performance of modern code: concurrent programs are difficult to design, write, reason about, debug, and tune. Concurrency can significantly affect the meaning of virtually every other construct in the language (beginning with the atomicity of assignment), and can affect the ability to invoke libraries. Despite this, most popular programming languages treat concurrency not as a language feature, but as a collection of external libraries that are often under-specified.

Considerable attention has been given, after the fact, to the specification of important concurrency libraries [Birrell et al. 1987; Gosling et al. 1996; Detlefs et al. 1998; Gurevich et al. 2000] to the point where one can usually determine what their behaviour should be under any implementation. Yet, even when the concurrency libraries are satisfactorily specified, the simple fact that they are libraries, and not features of the language, has undesirable consequences.

Many features can be provided, in principle, either as language features or as libraries: typical examples are memory management and exceptions. The advantage of having such features "in the language" is that the compiler can analyze them, and can therefore produce better code and warn programmers of potential and actual problems. In particular, the compiler can check for syntactically embedded invariants that would be difficult to extract from a collection of library calls. Moreover, programmers can more reliably state their intentions through a clear

Authors' address: Microsoft Research, Roger Needham Building, 7 J J Thomson Avenue, Cambridge CB3 0FB, United Kingdom.

This work was first presented at FOOL 9 – Foundations of Object-Oriented Languages, January 2002. A preliminary version of this paper appears in the proceedings of ECOOP 2002 – Object-Oriented Programming, LNCS 2374, June 2002.

syntax, and tools other than the compiler can more easily determine the programmers' intentions. Domain Specific Languages [Ramming 1997; Kamin 1997] are an extreme example of this linguistic approach: new ad-hoc languages are routinely proposed not to replace general-purpose language, but to facilitate domain-specific code analysis by the simple fact of expressing domain-related features as primitive language constructs.

We believe that concurrency should be a language feature and a part of language specifications. Serious attempts in this direction were made beginning in the 1970's with the concept of monitors [Hoare 1974] and the Occam language [INMOS Limited 1984] (based on Communicating Sequential Processes [Hoare 1985]). The general notion of monitors has become very popular, particularly in its current object-oriented form of threads and object-bound mutexes, but it has been provided at most as a veneer of syntactic sugar for optionally locking objects on method calls.

Many things have changed in concurrency since monitors were introduced. Communication has become more asynchronous, and concurrent computations have to be "orchestrated" on a larger scale. The concern is not as much with the efficient implementation and use of locks on a single processor or multiprocessor, but with the ability to handle asynchronous events without unnecessarily blocking clients for long periods, and without deadlocking. In other words, the focus is shifting from shared-memory concurrency to message- or event-oriented concurrency.

These new requirements deserve programming constructs that can handle well asynchronous communications and that are not shackled to the shared-memory approach. Despite the development of a large collection of design patterns [Lea 1999] and of many concurrent languages [America 1989; Agha et al. 1993; Reppy 1992; Pierce and Turner 2000; Philippsen 1995], only monitors have gained widespread acceptance as programming constructs.

An interesting new linguistic approach has emerged recently with Fournet and Gonthier's *join calculus* [1996; 2002], a process calculus well-suited to direct implementation in a distributed setting. Other languages, such as JoCaml [Conchon and Le Fessant 1999] and Funnel [Odersky 2000], combine similar ideas with the functional programming model. Here we propose an adaptation of join calculus ideas to an object-oriented language that has an existing threads-and-locks concurrency model. Itzstein and Kearney [2001] have recently described very similar extensions for Java.

1.2 Asynchronous Programming

Asynchronous events and message passing are increasingly used at all levels of software systems. At the lowest level, device drivers have to respond promptly to asynchronous device events, while being parsimonious on resource use. At the Graphical User Interface level, code and programming models are notoriously complex because of the asynchronous nature of user events; at the same time, users hate being blocked unnecessarily. At the wide-area network level, e.g. in collaborative applications, distributed workflow or web services, we are now experiencing similar problems and complexity because of the asynchronous nature and latencies of global communication.

In all these areas, we naturally find situations where there are many asynchronous messages to be handled concurrently, and where many threads are used to handle

them. Threads are still an expensive resource on most systems. However, if we can somewhat hide the use of messages and threads behind a language mechanism, then many options become possible. A compiler may transform some patterns of concurrency into state machines, optimize the use of queues, use lightweight threads when possible, avoid forking threads when not necessary, and use thread pools. All this is really possible only if one has a handle on the spectrum of "things that can happen": this handle can be given by a syntax for concurrent operations that can both hide and enable multiple implementation techniques.

Therefore, we aim to promote abstractions for asynchronous programming that are high-level, from the point of view of a programmer, and that enable low-level optimizations, from the point of view of a compiler and run-time systems. We propose an extension of the C♯ language with modern concurrency abstraction for asynchronous programming. In tune with the musical spirit of C♯ and with the "orchestration" of concurrent activities, we call this language Polyphonic C♯.[1]

1.3 C♯ and .NET

C♯ is a modern, type-safe, object-oriented programming language recently introduced by Microsoft as part of Visual Studio.NET [ECMA 2001]. C♯ programs run on top of the .NET Framework, which includes a multi-language execution engine and a rich collection of class libraries.

The .NET execution engine provides a multi-threaded execution environment with synchronization based on locks potentially associated with each heap-allocated object. The C♯ language includes a **lock** statement, which obtains the mutex associated with a given object during the execution of a block. In addition, the .NET libraries implement many traditional concurrency control primitives such as semaphores, mutexes and reader/writer locks, as well as an asynchronous programming model based on delegates.[2] The .NET Framework also provides higher-level infrastructure for building distributed applications and services, such as SOAP-based messaging and remote method call.

The concurrency and distribution mechanisms of the .NET Framework are powerful, but they are also undeniably complex. Quite apart from the bewildering array of primitives that are more or less 'baked in' to the infrastructure, there is something of a mismatch between the 1970s model of concurrency on a single machine (shared memory, threads, synchronization based on mutual exclusion) and the asynchronous, message-based style that one uses for programming web-based applications and services. C♯ therefore seems an ideal test-bed for our ideas on language support for concurrency in mainstream languages.

2. POLYPHONIC C# LANGUAGE OVERVIEW

This section describes the syntax and semantics of the new constructs in Polyphonic C♯ and then gives a more precise, though still informal, specification of the syntax.

[1] *Polyphony* is musical composition that uses simultaneous, largely independent, melodic parts, lines, or voices (Encarta World English Dictionary, Microsoft Corporation, 2001).

[2] An instance of a delegate class encapsulates an object and a method on that object with a particular signature. So a delegate is more than a C-style function pointer, but slightly less than a closure.

2.1 The Basic Idea

To C♯'s fairly conventional object-oriented programming model, Polyphonic C♯ adds just two new concepts: *asynchronous methods* and *chords*.

Asynchronous Methods. Conventional methods are synchronous, in the sense that the caller makes no progress until the callee completes. In Polyphonic C♯, if a method is declared *asynchronous* then any call to it is guaranteed to complete essentially immediately. Asynchronous methods never return a result (or throw an exception); they are declared by using the **async** keyword instead of **void**. Calling an asynchronous method is much like sending a message, or posting an event.

Since asynchronous methods have to return immediately, the behaviour of a method such as

```
async postEvent(EventInfo data) {
  // large method body
}
```

is the only thing it could reasonably be: the call returns immediately and 'large method body' is scheduled for execution in a different thread (either a new one spawned to service this call, or a worker from some pool). However, this kind of definition is actually rather rare in Polyphonic C♯. More commonly, asynchronous methods are defined using chords, as described below, and do not necessarily require new threads.

Chords. A *chord* (also called a 'synchronization pattern', or 'join pattern') consists of a header and a body. The header is a set of method declarations separated by '**&**'. The body is only executed once *all* the methods in the header have been called. Method calls are implicitly queued up until/unless there is a matching chord. Consider for example

```
public class Buffer {
  public string Get() & public async Put(string s) {
    return s;
  }
}
```

The code above defines a class *Buffer* with two instance methods, which are jointly defined in a single chord. Method **string** *Get*() is a synchronous method taking no arguments and returning a **string**. Method **async** *Put*(**string** *s*) is asynchronous (so returns no result) and takes a **string** argument.

If *buff* is a instance of *Buffer* and one calls the synchronous method *buff*.*Get*() then there are two possibilities:

—If there has previously been an unmatched call to *buff*.*Put*(*s*) (for some string *s*) then there is now a match, so the pending *Put*(*s*) is dequeued and the body of the chord runs, returning *s* to the caller of *buff*.*Get*().

—If there are no previous unmatched calls to *buff*.*Put*(.) then the call to *buff*.*Get*() blocks until another thread supplies a matching *Put*(.).

Conversely, on a call to the asynchronous method *buff*.*Put*(*s*), the caller never waits, but there are two possible behaviours with regard to other threads:

—If there has previously been an unmatched call to *buff*.*Get*() then there is now a match, so the pending call is dequeued and its associated blocked thread is awakened to run the body of the chord, which returns *s*.

—If there are no pending calls to *buff*.*Get*() then the call to *buff*.*Put*(*s*) is simply queued up until one arrives.

Exactly *which* pairs of calls are matched up is unspecified, so even a single-threaded program such as

```
Buffer buff = new Buffer();
buff.Put("blue");
buff.Put("sky");
Console.Write(buff.Get() + buff.Get());
```

is non-deterministic (printing either "bluesky" or "skyblue").[3]

Note that the implementation of *Buffer* does not involve spawning any threads: whenever the body of the chord runs, it does so in a preexisting thread (viz. the one that called *Get*()). The reader may at this point wonder what are the rules for deciding in which thread a body runs, or how we know to which method call the final value computed by the body will be returned. The answer is that in any given chord, at most one method may be synchronous. If there is such a method, then the body runs in the thread associated with a call to that method, and the value is returned to that call. Only if there is no such method (i.e. all the methods in the chord are asynchronous) does the body run in a new thread, and in that case there is no value to be returned.

It should also be pointed out that the *Buffer* code, trivial though it is, is thread-safe. The locking that is required (for example to prevent the argument to a single *Put* being returned to two distinct *Get*s) is generated automatically by the compiler. More precisely, deciding whether any chord is enabled by a call and, if so, removing the other pending calls from the queues and scheduling the body for execution is an atomic operation. Apart from this atomicity guarantee, however, there is *no* monitor-like mutual exclusion between chord bodies. Any mutual exclusion that is required must be programmed explicitly in terms of synchronization conditions in chord headers.

The *Buffer* example uses a single chord to define two methods. It is also possible (and common) to have multiple chords involving a given method. For example:

```
public class Buffer {
  public string Get() & public async Put(string s) {
    return s;
  }

  public string Get() & public async Put(int n) {
    return n.ToString();
  }
}
```

[3]In a real implementation the nondeterminism in this very simple example may be resolved statically, so different executions will always produce the same result; this is an allowable implementation.

Now we have defined one method for getting data out of the buffer, but two methods for putting it in (which happen to be distinguished by type rather than name). A call to *Get*() can synchronize with a call to either of the *Put*() methods. If there are queued calls to both *Put*()s, then which one synchronizes with a subsequent *Get*() is unspecified.

3. INFORMAL SPECIFICATION

3.1 Grammar

The syntactic extensions to the C♯ grammar [ECMA 2001, Appendix C] are very minor. We add a new keyword, **async**, and add it as an alternative *return-type*:

return-type ::= *type* | **void** | **async**

This allows methods, delegates and interface methods to be declared asynchronous. In *class-member-declaration*s, we replace *method-declaration* with *chord-declaration*:

chord-declaration ::=
  method-header [**&** *method-header*]* *body*
method-header ::=
  attributes modifiers return-type member-name(*formals*)

We call a chord declaration *trivial* if it declares a single, synchronous method (i.e. it is a standard C♯ method declaration).

3.2 Well-Formedness

Extended classes are subject to a number of well-formedness conditions:

—Within a single *method-header*:

(1) If *return-type* is **async** then the formal parameter list *formals* may not contain any **ref** or **out** parameter modifier.[4]

—Within a single *chord-declaration*:

(2) At most one *method-header* may have a non-**async** *return-type*.
(3) If the chord has a *method-header* with *return-type type*, then *body* may use **return** statements with *type* expressions, otherwise *body* may use empty **return** statements.
(4) All the *formals* appearing in *method-header*s must have distinct identifiers.
(5) Two *method-header*s may not have both the same *member-name* and the same argument type signature.
(6) The *method-header*s must either all declare instance methods or all declare static methods.

—Within a particular class:

(7) All *method-header*s with the same *member-name* and argument type signature must have the same *return-type* and identical sets of *attributes* and *modifiers*.

[4]Neither **ref** nor **out** parameters make sense for asynchronous messages, since they are both passed as addresses of locals in a stack frame that may have disappeared when the message is processed.

(8) If it is a value class (**struct**), then only static methods may appear in non-trivial chords.
(9) If any *chord-declaration* includes a virtual method m with the **override** modifier[5], then any method n that appears in a chord with m *in the superclass containing the overridden definition of* m must also be overridden in the subclass.

Most of these conditions are fairly straightforward, though Conditions 2 and 9 deserve some further comment.

Condition 9 provides a conservative, but simple, sanity check when refining a class that contains chords since, in general, implementation inheritance and concurrency do not mix well [Matsuoka and Yonezawa 1993] (see Fournet et al. [2000] for a discussion of "inheritance anomalies" in the context of the join calculus). Our approach here is to enforce a separation of these two concerns: a series of chords must be syntactically local to a class or a subclass declaration; when methods are overridden, all their chords must also be completely overridden. If one takes the view that the implementation of a given method consists of all the synchronization and bodies of all the chords in which it appears then our inheritance restriction seems not unreasonable, since in (illegal) code such as

```
class C {
    virtual void f() & virtual async g() { /* body1 */ }
    virtual void f() & virtual async h() { /* body2 */ }
}

class D : C {
    override async g() { /* body3 */ }
}
```

one would, by overriding $g()$, have also 'half' overridden $f()$.

More pragmatically, removing the restriction on inheritance makes it all too easy to introduce inadvertent deadlock (or 'async leakage'). If the code above were legal, then code written to expect instances of class C that makes matching calls to $f()$ and $g()$ would fail to work when passed an instance of D—all the calls to $g()$ would cause body3 to run and all the calls to $f()$ would deadlock.

Note that the inheritance restriction means that declarations such as

```
virtual void f() & private async g() { /* body1 */ }
```

are incorrect: declaring just one of $f()$ and $g()$ to be **virtual** makes no sense (and is flagged as an error by our compiler), as overriding one requires the other to be overridden too. It is also worth observing that there is a transitive closure operation implicit in our inheritance restriction: if $f()$ is overridden and joined with $g()$ then because $g()$ must be overridden, so must any method $h()$ that is joined with $g()$ and so on.

[5]In C♯, methods that are intended to be overridable in subclasses are explicitly marked as such by use of the **virtual** modifier, whilst methods that are intended to override ones inherited from a superclass must explicitly say so with the **override** modifier.

It is possible to devise more complex and permissive rules for overriding. Our current rule has the advantage of simplicity, but we refer the reader to Fournet et al. [2000] for a more thorough study of inheritance and concurrency in the join calculus. In that paper, classes are collections of (partial) synchronization patterns, which can be combined and transformed using a few inheritance operators. As usual, objects can then be created by instantiating classes, and their synchronization patterns are not extensible. The composition of classes is controlled by a sophisticated typing discipline that prevents "message not understood" errors at runtime.

Well-formedness Condition 2 above is also justified by a potentially bad interaction between existing C♯ features and the pure join calculus. Allowing more than one synchronous call to appear in a single chord would give a potentially useful *rendezvous* facility (provided one also added syntax allowing results to be returned to particular calls). For example, instances of the following class

```
class RendezVous {
  public int f(int i) & public int g(int j) {
    return j to f;
    return i to g;
  }
}
```

would match pairs of calls to f and g, which then exchange their values and proceed. However, one would also have to decide in which of the blocked threads the body should run, and this choice is generally observable. If this were only because thread identities can be obtained and checked for equality, the problem would be fairly academic. But, in C♯, the choice of thread could make a significant difference to the behaviour of the program—due to reentrant locks, stack-based security and thread-local variables—thus making **&** 'very' non-commutative.

Of course, it is not hard to program explicitly the rendezvous above in Polyphonic C♯:

```
class RendezVous {
  class Thunk {
    int wait() & async reply(int j) { return j; }
  }
  public int f(int i) {
    Thunk t = new Thunk();
    af(i,t);
    return t.wait();
  }
  private async af(int i,Thunk t) & public int g(int j) {
    t.reply(j); // returning to f
    return i;   // returning to g
  }
}
```

For each call to f, we create an instance of the auxiliary class *Thunk*, in order to *wait* for an asynchronous *reply* message, which is sent after synchronization with some g.

3.3 Typing Issues

We treat **async** as a subtype of **void** and allow *covariant return types* just in the case of these two (pseudo)types. Thus

—an **async** method may override a **void** one,

—a **void** delegate may be created from an **async** method, and

—an **async** method may implement a **void** method in an interface

but not conversely. This design makes intuitive sense (an **async** method *is* a **void** one, but has the extra property of returning 'immediately') and also maximizes compatibility with existing C♯ code (superclasses, interfaces and delegate definitions) making use of **void**.

4. PROGRAMMING IN POLYPHONIC C♯

Having introduced the language, we now show how it may be used to address a range of concurrent programming problems.

4.1 A Simple Cell Class

We start with an implementation of a simple one-place cell class. Cells have two public synchronous methods: **void** *Put*(**object** *o*) and **object** *Get*(). A call to *Put* blocks until the cell is empty and then fills the cell with its argument. A call to *Get* blocks until the cell is full and then removes and returns its contents:

```
public class OneCell {
  public OneCell() {
    empty();
  }
  public void Put(object o) & private async empty() {
    contains(o);
  }
  public object Get() & private async contains(object o) {
    empty();
    return o;
  }
}
```

In addition to the two public methods, the class uses two private asynchronous methods, *empty*() and *contains*(**object** *o*), to carry the state of cells. There is a simple declarative reading of the constructor and the two chords that explains how this works:

Constructor. When a cell is created, it is initially *empty*().

Put-chord. If we *Put* an **object** *o* into a cell that is *empty*() then the cell subsequently *contains*(*o*).

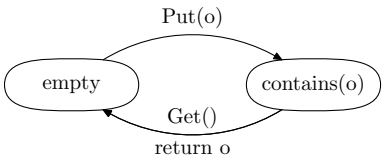
Get-chord. If we *Get*() the contents of a cell that *contains* an **object** *o* then the cell becomes *empty*() and the returned value is *o*.

Implicitly. In all other cases, *Put*s and *Get*s wait.

The technique of using private asynchronous methods (rather than fields) to carry state is very common in Polyphonic C♯. Observe that the constructor establishes, and every body in class *OneCell* preserves, a simple and easily verified invariant:

> There is always exactly one pending asynchronous method call: either *empty*(), or *contains*(*o*) for some **object** *o*.

(In contrast there may be an arbitrary number of client threads blocked with pending calls to *Put* or *Get*, or even concurrently running statement **return** *o* within the last body.) Hence one can also read the class definition as a direct specification of an automaton:



4.2 Reader-Writer Locks

As a more realistic example of the use of asynchronous methods to carry state and chords to synchronize access to that state, we now consider the classic problem of protecting a shared mutable resource with a multiple-reader, single-writer lock. Clients each request, and then release, either shared access or exclusive access, using the corresponding public methods *Shared*, *ReleaseShared*, *Exclusive*, and *ReleaseExclusive*. Requests for shared access block until no other client has exclusive access, whilst requests for exclusive access block until no other client has any access. A canonical solution to this problem using traditional concurrency primitives in Modula 3 is given by Birrell [1989]; using Polyphonic C♯, it can be written with just five chords:

```
class ReaderWriter
{
  ReaderWriter() { idle(); }

  public void Shared() & async idle() { s(1); }
  public void Shared() & async s(int n) { s(n+1); }
  public void ReleaseShared() & async s(int n) {
    if (n == 1) idle(); else s(n-1);
  }
  public void Exclusive() & async idle() {}
  public void ReleaseExclusive() { idle(); }
}
```

Provided that every release follows the corresponding request, the invariant is that the state of the lock (no message, a single message *idle*(), or a single message $s(n)$ with $n > 0$) matches the kind and number of threads currently holding the lock (an exclusive thread, no thread, or n sharing threads).

In case there is at most one message pending on a given private method, it is a matter of choice whether to use private fields in the object or parameters in

the private message. In the example above, n is relevant only when there is an $s()$ message present. Nonetheless, we could write instead the following equivalent code:

```
class ReaderWriterPrivate
{
  ReaderWriter() { idle(); }
  private int n = 0; // protected by s()

  public void Shared() & async idle() { n=1; s(); }
  public void Shared() & async s() { n++; s(); }
  public void ReleaseShared() & async s() {
    if (--n == 0) idle(); else s();
  }
  public void Exclusive() & async idle() {}
  public void ReleaseExclusive() { idle(); }
}
```

Our implementation and the underlying operating system scheduler provide only basic fairness properties—for instance, if there are enough pending calls in a polyphonic object to match a chord, then at least one chord body eventually runs. Hence, it is often useful to program explicitly some additional application-specific fairness or priority. For example, with the code above, writers may not be able to acquire an exclusive lock as long as new readers keep acquiring a shared lock. We further refine this code to implement a particular fairness policy between readers and writers: when there are pending writers, at least one writer will acquire the lock after all current readers release it. To this end, we add extra shared states: $t()$, in which we do not accept new readers, and *idleExclusive*(), in which we provide the exclusive lock to a previously-selected thread:

```
class ReaderWriterFair
{
   ... // same content as in ReaderWriterPrivate, plus:

  public void ReleaseShared() & async t() {
    if (--n == 0) idleExclusive(); else t();
  }
  public void Exclusive() & async s() { t(); wait(); }
  void wait() & async idleExclusive() {}
}
```

4.3 Combining Asynchronous Messages

The external interface of a server that uses message-passing will typically consist of asynchronous methods, each of which takes as arguments both the parameters for a request *and* somewhere to send the final result or notification that the request has been serviced. For example, using delegates as callbacks, a service taking a string argument and returning an integer might look like:

```
public delegate async IntCallback(int result);

public class Service {
  public async Request(string arg, IntCallback cb) {
    int r;
     ...      // do some work
    cb(r); // send the result back
  }
}
```

A common client-side pattern then involves making several concurrent asynchronous requests and later blocking until *all* of them have completed. This may be programmed as follows:

```
class Join2 {
  public IntCallback firstcb;
  public IntCallback secondcb;
  public Join2() {
    firstcb  = new IntCallback(first);
    secondcb = new IntCallback(second);
  }
  public void wait(out int i, out int j)
  & async first(int fst)
  & async second(int snd) {
    i = fst; j = snd;
  }
}

class Client {
  public static void Main(string[] args) {
    Service s1 = ...;
    Service s2 = ...;
    Join2 x = new Join2();
    s1.Request(args[0], x.firstcb);
    s2.Request(args[1], x.secondcb);
     ...                    // do something useful in the meantime...
    int i,j;
    x.wait(out i, out j); // wait for both results to come back
     ...                    // do something with them
  }
}
```

The call to $x.wait(i,j)$ will block until/unless both of the services have replied by invoking their respective callbacks on x. Once that has happened, the two results will be assigned to i and j and the client will proceed. Generalizing *Join2* (which, of course, naturally belongs in a general-purpose library) to an arbitrary number of simultaneous calls, or defining classes that wait for conditions such as 'at least 3 out of 5 calls have completed' is straightforward.

4.4 Active Objects

Actors [Hewitt 1977; Agha 1990] model concurrency in terms of active agents that communicate by asynchronous message passing. Based on this idea, a number of programming languages, such as ABCL/1 [Yonezawa 1990], have been designed around the principle of unifying the notions of process and object to yield *active objects*. A simple version of this model gives each active object its own thread of control, which sequentially processes asynchronous messages received from other such objects. One way to express this pattern in Polyphonic C♯ is via inheritance from an abstract base class:

```
public abstract class ActiveObject {
  protected bool done;

  abstract protected void ProcessMessage();

  public ActiveObject () {
    done = false;
    mainLoop();
  }

  async mainLoop() {
    while (!done) {
      ProcessMessage();
    }
  }
}
```

The constructor of *ActiveObject* calls the asynchronous method *mainLoop*(), which spawns a new message-handling thread for that object. Subclasses of *ActiveObject* then define chords for each message to synchronize with a call to *ProcessMessage*(). Here, for example, is a skeleton of an active object that multicasts stock quote messages to a list of clients:

```
public class StockServer : ActiveObject {
  private ArrayList clients = new ArrayList();

  public async AddClient(Client c) // add new client
  & override protected void ProcessMessage() {
    clients.Add(c);
  }
  public async WireQuote(Quote q) // get new quote off wire
  & override protected void ProcessMessage() {
    foreach (Client c in clients) {
      c.UpdateQuote(q);  // and send to all clients
    }
  }

  public async CloseDown() // request to terminate
```

```
    & override protected void ProcessMessage() {
      done = true;
    }
  }
```

Note that access to *done* and *clients* need not be protected by a lock, since only the message-handling thread accesses them. Also, one might attempt to share the *CloseDown*() behaviour amongst all active objects by moving the last chord to the superclass and making *ProcessMessage*() **virtual** instead of **abstract**, but this would be caught at compile-time as a violation of the inheritance restriction of Section 3.2.

4.5 Custom Schedulers

In Polyphonic C♯, we have to both coexist with and build upon the existing threading model. Because these threads are relatively expensive, and are the holders of locks, C♯ programmers often need explicit control over thread usage. In such cases, Polyphonic C♯ is a convenient way to write what amount to custom schedulers for a particular application.

To illustrate this point, we present an example in which we dynamically schedule series of related calls in large batches, to favour locality, in the spirit of the staged computation server of Larus and Parkes [2001].

Assume the class *Heavy* encapsulates access to expensive resources, such as files or remote connections. Each client first allocates an instance of class *Heavy*, then performs a series of calls to *Work*, and eventually calls *Close* to release the resource. Calls to the constructor *Heavy*(*resourceId*) are assumed to be potentially blocking and relatively expensive.

```
class Heavy {
  public Heavy (int resourceId) { /* so slow! */ }
  public int Work(int request) { /* relatively fast */ }
  public void Close () { ... }
}
```

The class below implements our scheduler. For each resource q, an instance of class *Burst* provides a front-end that attempts to organize calls into long series that share the cost of *Heavy*(q). A burst can be in two states, represented by either *idle*() or *open*(). The state is initially idle. When a first thread actually tries to use the resource, the state becomes *open*(), and the thread calls *Work*(p) on the result of a potentially-blocking *Heavy*(q) call. As long as the state is open, subsequent callers are queued-up. When the first thread completes its *Work*, and before closing the *Heavy* resource, it also calls *Work* on behalf of any pending calls, resuming their threads with the respective results. Meanwhile, the state is still open, and new threads may be queued-up. As long as there are pending calls, they are similarly processed; otherwise, the state becomes idle again. As in Section 3.2, the auxiliary class *Thunk* is used to block each queued-up thread and resume it with an asynchronous message carrying the result r.

```
class Burst {
    int others = 0;
    int q;
    public Burst(int q) { this.q = q; idle (); }

    public int Work(int p) & async idle() {
      open();
      Heavy h = new Heavy(q);
      int r = h.Work(p);
      helpful (h); // any delayed threads?
      h.Close();
      return r;
    }
    public int Work(int p) & async open() {
      others++; open();
      Thunk t = new Thunk(); delayed(t,p);
      return t.Wait(); // usually blocking
    }
    void helpful(Heavy h) & async open() {
      if (others == 0) idle();
      else {
          int batch = others; others = 0;
          open();
          while(batch-- > 0) extraWork(h);
          helpful (h); // newly−delayed threads?
      }
    }
    void extraWork(Heavy h) & async delayed(Thunk t,int p) {
      t.Done(h.Work(p));
    }
}

class Thunk {
    public int Wait() & public async Done(int r) {
        return r;
    }
}
```

We have written simulations that, unsurprisingly, exhibit a large speedup when numerous client threads call *Burst* rather than independently calling *Heavy*.

5. IMPLEMENTATION

This section describes the implementation of chords using lower-level concurrency primitives. The compilation process is best explained as a translation from a polyphonic class to a plain C♯ class. The resulting class has the same name and signature as the source class (after mapping **async** to **void**), and also has private state and methods to deal with synchronization.

5.1 Synchronization and State Automata

In the implementation of a polyphonic class, each method body combines two kinds of code, corresponding to the synchronization of polyphonic method calls (generated from the chord headers) and to their actual computation (copied from the chord bodies), respectively.

We now describe how the synchronization code is generated from a set of chords. Since synchronization is statically defined by those chords, we can efficiently compile it down to a state automaton. This is the approach initially described by Le Fessant and Maranget [1998], though our implementation does not construct explicit state machines.

The *synchronization state* consists of the pending calls for all methods that occur in (non-trivial) chords, that is, threads for regular methods and messages for asynchronous methods. However, synchronization depends only on the presence or absence of pending calls to each method; the number of calls, the actual parameters and the calling contexts become relevant only after a chord is fired. Hence, the whole synchronization state can be summarized in a bitmap, with a single bit that records the presence of (one or more) pending calls, for each method appearing in a least one chord. Accordingly, every chord declaration is represented as a constant bitmap with a bit set for every method appearing in that chord, and the synchronization code checks whether a chord can be fired by testing the synchronization bitmask against constant bitmasks.

Performance considerations. Ideally, the cost of polyphonic method calls should be similar to that of regular method calls unless the call blocks waiting for **async** messages—in that case, we cannot avoid paying the rather high cost of dynamic thread scheduling.

When an asynchronous method is called, it performs a small amount of computation on the caller thread before returning.

When a synchronous method is called, the critical path to optimize is the one in which, for at least one chord, all complementary asynchronous messages are already present. In that case, the synchronization code retrieves the content of the complementary messages, updates the synchronization state, and immediately proceeds with the method body. Otherwise, the thread must be suspended, and the cost of running our synchronization code is likely to be small as compared to lower-level context-switching and scheduling.

Firing a completely asynchronous chord is always comparatively expensive since it involves spawning a new thread. Hence, when an asynchronous message arrives, it makes sense to check for matches with synchronous chords first. We have also tried lowering the cost of asynchronous chords by using .NET's *thread pool* mechanism rather than simply spawning a fresh system thread every time. The limits and scheduling policy of the thread pool are problematic for some applications, however, so we have now reverted to creating fresh threads (a future refinement may be to use attributes[6] to allow programmer control over thread creation policy).

[6]Attributes are a standardized, declarative way of adding custom metadata to .NET programs. Code-manipulating tools and libraries, such as compilers, debuggers or the object serialization libraries can then use attribute information to vary their behaviour.

Low-level Concurrency. The code handling the chords must be thread-safe, for all source code in the class. To ensure this, we use a single, auxiliary lock protecting the private synchronization state of each object.[7] Locking occurs only briefly for each incoming call, and involves a separate lock for each polyphonic object, so we expect contention to be rare compared with more typical C♯ programs, which hold object locks during non-trivial computations.[8]

This lock is independent of the regular object lock, which may be used as usual to protect the rest of the state and prevent race conditions while executing chord bodies.

5.2 The Translation

We now present, by means of a simple example, the details of the translation of Polyphonic C♯ into ordinary C♯. The translation presented here is actually an abstraction of those we have implemented: for didactic purposes, we modularize the translated code by introducing auxiliary classes for queues and bitmasks, whereas our current implementation mostly inlines the code contained in these classes.

Supporting Classes. The following value class (structure) provides operations on bitmasks:

```
struct BitMask {
  private int v; // = 0;
  public void set(int m) { v |= m; }
  public void clear(int m) { v &= ~m; }
  public bool match(int m) { return (~v & m)==0; }
}
```

Next, we define the classes that represent message queues. To every asynchronous method appearing in a chord involving more than one method, the compiler associates a queue of pending messages, with an *empty* property for testing its state and two methods, *add* and *get*, for enqueueing and dequeueing entries. The implementation of each queue depends on the message contents (and, potentially, on compiler-deduced invariants); it does not necessarily use an actual queue.

A simple case is that of single-argument asynchronous messages (here, **int** messages); these generate a thin wrapper on top of the standard queue library:[9]

```
class intQ {
  private Queue q;
  public intQ() {q = new Queue(); }
  public void add(int i) { q.Enqueue(i); }
  public int get() {return (int) q.Dequeue(); }
  public bool empty {get{return q.Count == 0;}}
}
```

[7] We actually use the regular object lock for one of the asynchronous queues, if a suitable one is free; otherwise we allocate a fresh object just for its lock.

[8] On a multiprocessor, using a spinlock may be appropriate here.

[9] Readers unfamiliar with C♯ may be worried by the definition of *empty*. This is a (read-only) *property*—a parameterless method that can be called using field-like syntax.

Another important case is that of empty (no argument) messages. Queues for such messages are implemented as a simple counter.

```
class voidQ {
  private int n;
  public voidQ() { n = 0; }
  public void add() { n++; }
  public void get() { n--; }
  public bool empty {get{ return n==0; }}
}
```

Finally, for synchronous methods, we need classes implementing queues of waiting threads. As with message queues, there is a uniform interface and a choice of several implementations. Method *yield* is called to store the current thread in the queue and await additional messages; it assumes the thread holds some private lock on a polyphonic object, and releases that lock while waiting. Conversely, method *wakeup* is called to wake up a thread in the queue; it immediately returns and does not otherwise affect the caller thread.

The first version of our compiler managed thread queues explicitly and used the *Thread.Sleep*() and *Thread.Interrupt*() methods of the .NET Framework to block and resume threads, using the following implementation:

```
class threadQ {
  private Queue q;
  private bool interrupted = false;
  public threadQ() { q = new Queue(); }
  public bool empty {get{ return (q.Count == 0); }}
  public void yield(object myCurrentLock) {
    q.Enqueue(Thread.CurrentThread);
    Monitor.Exit(myCurrentLock);
    try {
      Thread.Sleep(Timeout.Infinite);
    } catch (ThreadInterruptedException) {}
    Monitor.Enter(myCurrentLock);
    q.Dequeue();
    interrupted = false;
  }
  public void wakeup() {
    if (!interrupted) {
      ((Thread) q.Peek()).Interrupt();
      interrupted = true;
    }
  }
}
```

The specification of monitors guarantees that an interrupt on a non-sleeping thread does not happen until the thread actually does enter a sleeping or waiting state, hence it *is* correct to release the lock before entering the **try catch** statement. As the thread awakens in the **catch** clause, it re-acquires the lock and then pops its

queue (the thread that is dequeued and discarded is always the current thread). The *interrupted* flag is used to ensure that the thread at the head of the queue is only interrupted once.

The interruption-based implementation of thread queues was the most efficient on the 1.0 version of .NET, though it had some disadvantages (see Section 6). The 1.1 release significantly improved the performance of the *Monitor.Wait*(), *Monitor.Pulse*() and *Monitor.PulseAll*() methods[10], so we now implement thread queues using the built-in support for waiting and notification instead:

```
class threadQ {
  private bool signalled = false;
  private int count = 0;
  public bool empty {get{ return (count == 0); }}

  public void yield(object myCurrentLock) {
    count++;
    Monitor.Exit(myCurrentLock);
    lock(this) {
      while (!signalled ) {
        Monitor.Wait(this);
      }
      signalled = false;
    }
    Monitor.Enter(myCurrentLock);
    count--;
  }

  public void wakeup() {
    lock(this) {
      if (! signalled ) {
        signalled = true;
        Monitor.Pulse(this);
      }
    }
  }
}
```

The queue of threads blocked on a call to a synchronous method is now implemented as the wait queue of the *threadQ* object itself, which essentially implements a binary semaphore.

Generated Synchronization Code. Figure 1 shows a simple polyphonic class *Token* (from Section 4.5, though with the addition of a parameter passed to and returned from the *Grab* method) and its translation into ordinary C♯, making use of the auxiliary classes defined above. *Token* implements an n-token lock. It has a regular

[10]These operations have the same behaviour as *Object.wait*, *Object.notify* and *Object.notifyAll* in Java.

```
class Token {
  public Token(int initial_tokens) {
    for (int i = 0; i < initial_tokens ; i++) Release();
  }
  public int Grab(int id) & public async Release() {
    return id;
  }
}
```

```
class Token {
  private const int mGrab = 1 << 0;
  private const int mRelease = 1 << 1;
  private threadQ GrabQ = new threadQ();
  private voidQ ReleaseQ = new voidQ();

  private const int mGrabRelease = mGrab | mRelease;
  private BitMask s = new BitMask();
  private object mlock = ReleaseQ;

  private void scan() {
    if (s.match(mGrabRelease)) {GrabQ.wakeup(); return;}
  }
  public Token(int initial_tokens) {
    for (int i = 0; i < initial_tokens ; i++) Release();
  }
  [OneWay] public void Release() {
    lock(mlock) {
      ReleaseQ.add();
      if (!s.match(mRelease)) {
      s.set(mRelease);
      scan(); }}
  }
  public int Grab(int id) {
    Monitor.Enter(mlock);
    if (!s.match(mGrab)) goto now;
  later:
    GrabQ.yield(mlock); if (GrabQ.empty) s.clear(mGrab);
  now:
    if (s.match(mRelease)) {
      ReleaseQ.get(); if (ReleaseQ.empty) s.clear(mRelease);
      scan();
      Monitor.Exit(mlock);
      {
      return id; // source code for the chord
      }
    }else{
      s.set(mGrab); goto later; }}
}
```

Fig. 1. The *Token* class and its translation

synchronous method, an asynchronous method, and a single chord that synchronizes the two.

We now describe what is happening in the translations of the two methods:

Code for Release. After taking the chord lock, we add the message to the queue and, unless there were already messages stored in *ReleaseQ*, we update the mask and scan for active chords.

In a larger class with chords that do not involve *Release*, the *scan*() statement could be usefully inlined and specialized: we only need to test patterns where **async** *Release*() appears; besides, we know that the *mRelease* bit is set.

The use of OneWay. The reader unfamiliar with C♯ may wonder why the translation of the *Release*() method is prefixed with '[*OneWay*]'. This is an attribute that indicates to the .NET infrastructure that where appropriate (e.g. when calling between different machines) calls of *Release*() should be genuinely non-blocking. The translation adds this attribute to all asynchronous methods.

Code for Grab. After taking the chord lock, we first check whether there are already deferred *Grab*s stored in *GrabQ*. If so, this call cannot proceed for now so we enqueue the current thread and will retry later.

Otherwise, we check whether there is at least one pending *Release* message to complete the chord **int** *Grab*(**int** *id*) **&** **async** *Release*(). If so, we select this chord for immediate execution; otherwise we update the mask to record the presence of deferred *Grab*s, enqueue the current thread and will retry later. (In classes with multiple patterns for *Grab*, we would perform a series of tests for each potential chord.) Notice that it is always safe to retry, independently of the synchronization state.

Once a chord is selected, we still have to update *ReleaseQ* and the mask. (Here, we have no asynchronous parameters; more generally, we would remove them from the queue and bind them to local variables.) Next, we check whether there are still enough messages to awaken another thread, as discussed below. Finally, we release the lock and enter the block associated with the selected chord.

Why rescanning?. One may wonder why we systematically call *scan*() after selecting a chord for immediate execution (just before releasing the lock and executing the guarded block). In our simple example, this is unnecessary whenever we already know that this was the last *scan*() call or the last *Release*() message. In general, however, this may be required to prevent deadlocks. Consider for instance the polyphonic class

```
class Foo {
  void m1() & async s() & async t() {...}
  void m2() & async s() {...}
  void m3() & async t() {...}
}
```

and the following global execution trace, with four threads running in parallel:

Thread 1. calls *m1*() and blocks.

Thread 2. calls *m2*() and blocks.

Thread 0. calls $t()$ then $s()$, awaking Thread 1

Thread 3. calls *m3*() and succeeds, consuming $t()$.

Thread 1. retries *m1*() and blocks again.

Observe that, as Thread 0 awakes Thread 1, there is no guarantee that Thread 1 runs at once—on the contrary, Thread 0 typically keeps running until it is preempted, while Thread 1 is added to the queue of executable threads. In our case, there is a race condition between Thread 1 and Thread 3 to consume $t()$. Thread 3 preempts Thread 1, which is left with a single message $s()$ and blocks again. In the final state, only Thread 2 can consume $s()$ but if no other thread awakens it, we will have a deadlock.

Accordingly, in our implementation, the synchronization code in Thread 3 performs an additional *scan*() that awakes Thread 2 in such unfortunate cases. (In many special cases, the final *scan*() can safely be omitted, but identifying these cases would complicate the translation.)

Deadlock Freedom. We now sketch a proof that our translation does not introduce deadlocks. (Of course, calls involving a chord that is never fired may be deadlocked, and our translation must implement those deadlocks.)

We say that an object is *active* when there are enough calls in the queues to trigger one of its patterns; assuming a fair scheduling of runnable threads, we show that active states are transient. We prove the invariant: *when an object is active, at least one thread on top of a queue is scheduled for execution and can succeed.*

—After *scan*(), the invariant always holds.

—An object becomes active when an asynchronous message is received, and this always triggers a scan.

—A thread whose polyphonic call succeeds (and thus consumes asynchronous messages) also triggers a scan.

When the algorithm awakes a thread, it is guaranteed that this thread may succeed if immediately scheduled, but not that it will necessarily succeed.

Fully Asynchronous Chords. To complete the description of our implementation, we describe the compilation of fully asynchronous chords. When such chords are fired, there is no thread at hand to execute their body, so a new thread must be created.

To illustrate this case, assume the class *Token* also contains the asynchronous method declaration

```
public async Live(string s,int id) {
  Grab(id); Release();
  Console.WriteLine(s);
}
```

The generated code is verbose but straightforward:

```
private class liveThunk {
  string s; int id;
  Token parent;
```

```
    public liveThunk(Token parent, string s, int id) {
      this.s = s; this.id = id;
      this.parent = parent;
    }
    public void run() {
      parent.liveBody(s,id);
    }
  }

  private void liveBody(string s, int id) {
    Grab(id); Release(); // async chord body code
    Console.WriteLine(s);
  }

  public void Live(string s,int id) {
    liveThunk th = new liveThunk(this,s,id);
    ThreadStart d = new ThreadStart(th.run);
    (new Thread(d)).Start();
  }
```

The new thread starts by invoking a delegate to the *run* method on a fresh instance of an auxiliary class *liveThunk*. The *run* method calls the *liveBody* method on the hosting *Token* object, passing the arguments to the original call to *Live*.

More generally, for a chord containing several asynchronous methods, code analogous to that in the *Live* method above occurs instead of *mQ.wakeup*() to fire the pattern in method *scan*().

6. DISCUSSION AND FUTURE WORK

6.1 Implementations and Samples

We have two prototype implementations of Polyphonic C♯. The first is a lightweight, source-to-source translator written in ML. This has proven invaluable in explaining the language to others, and is straightforward to modify and maintain, though it does not cope with the full language. As our initial experiences using Polyphonic C♯ were positive, we have recently built a more robust, full-featured and maintainable implementation on top of an 'experimentation-friendly' C♯-in-C♯ compiler being developed by another group within Microsoft.

We have written a number of non-trivial samples in Polyphonic C♯, including some web combinators along the lines of Cardelli and Davies [1999], an animated version of the dining philosophers, a distributed stock-dealing simulation built on .NET's remoting infrastructure[11], a multi-threaded client for the TerraServer web service [Barclay et al. 2000; Barclay et al. 2002], and a solution [Benton 2003] to the "Santa Claus" problem [Trono 1994; Ben-Ari 1998]. In all these cases, we could rapidly, correctly and concisely express the intended concurrency and synchronization. When interfacing with libraries, however, we sometimes had to

[11]Remoting provides remote method call over TCP (binary) or HTTP (SOAP).

write thin wrappers providing a polyphonic interface to code written in a different style (for example, the auto-generated proxy classes for web services).

It is interesting that we have been able to implement our high-level concurrency primitives in two rather different ways as the performance tradeoffs of the underlying runtime have changed (see below). The original technique of implementing thread queues using interruption had some drawbacks, however. The most obvious is that it partially 'uses up' the ability to interrupt threads for other reasons. If one tries to interrupt a thread during a wait in a chord then the interrupt will effectively be ignored; if the interrupt is delivered during a wait in user code, then an exception will be thrown. In many situations where one might traditionally use thread interruption, such as responding the the user cancelling a long running blocking operation, programming the cancellation behaviour more explicitly in Polyphonic C♯ is straightforward, and arguably preferable. Nevertheless, this is an area in which our first implementation was less compatible with the pre-existing concurrency model than it could be. A further disadvantage is that .NET code that uses thread interruption requires slightly higher security permissions to run. Our current compilation scheme, using $Monitor.Wait()$ and $Monitor.Pulse()$ is much more satisfactory.

6.2 Performance

In practice, overall performance was not an issue with any of our samples. Our implementation of concurrency abstraction entirely relies on the .NET framework, and largely reflects its general performance. Besides, in the presence of remote messaging, the costs of local synchronization become negligible.

We have, however, run a small number of single-machine micro-benchmarks to estimate the costs of our primitives and to compare the efficiency of small samples coded in the 'natural' style in Polyphonic C♯ with their 'natural' C♯ equivalents. Our quantitative results should be treated with caution: small changes in the code or test environment can yield very different results on such small tests.

The performance figures (in thousands of operations per second) for each test are shown in Figure 2. All figures refer to the .NET Framework version 1.1. The single-processor numbers were collected using Windows XP Professional SP1 with an Athlon 1500+ processor and 512MB RAM; The dual-processor numbers were collected using Windows Server 2003 on a machine with two 730Mhz Pentium III processors and 640MB RAM.

Calling a method that is defined in a chord involves at least acquiring and releasing a lock and some bitmap operations. It may also involve allocation of garbage-collected objects to hold method arguments and comparatively expensive calls to the .NET queue classes. The first seven lines of the table indicate the speed of different forms of void, parameterless non-polyphonic method calls with an empty body, plus a call with a string argument, as a baseline for assessing these costs. (Adding an integer parameter yields much the same times as the parameterless cases, so we omit those figures.) The four 'instance call $s()$ consuming' lines give the performance of a call to s defined by **void** $s()$ **& async** $a()$ {}, **void** $s()$ **& async** $a1()$ **& async** $a2()$ {}, **void** $s()$ **& async** a(**int** v) {}, or **void** $s()$ **& async** a(**string** v) {}, respectively, in case synchronization always succeeds

Single-Processor Benchmark	Operations per second (thousands) Polyphonic	Non-polyphonic
virtual $s()$		143,000
instance $s()$		333,000
static $s()$		333,000
synchronized virtual $s()$		15,600
synchronized instance $s()$		15,900
synchronized static $s()$		15,600
synchronized instance s(string)		14,400
instance $s()$ consuming $a()$	12,000	
instance $s()$ consuming $a1()$ and $a2()$	11,000	
instance $s()$ consuming a(**int**)	5,260	
instance $s()$ consuming a(**string**)	4,050	
instance $a()$ queued	18,200	
instance a(**int**) queued	1,660	
instance a(**string**) queued	5,260	
instance $s()$ consuming and sending $a()$	6,540	
instance $s()$ consuming and sending $a1()$ and $a2()$	3,950	
instance $s()$ consuming and sending a(**int**)	2,810	
instance $s()$ consuming and sending a(**string**)	2,720	
ping pong	115	240
bounded buffer size=100, 1 producer, 1 consumer	682	115
bounded buffer size=100, 2 producers, 2 consumers	423	118

Dual-Processor Benchmark	Operations per second (thousands) Polyphonic	Non-polyphonic
ping pong	66	70
bounded buffer size=100, 1 producer, 1 consumer	288	250
bounded buffer size=100, 2 producers, 2 consumers	125	42

Fig. 2. Performance on micro-benchmarks

(that is, with plenty of messages already present on on a, $a1$, $a2$)[12]. Conversely, the three 'instance $a(\dots)$' lines give the performance of calling an asynchronous method with no parameter, an **int** parameter, and a **string** parameter respectively, in case synchronization fails and the message is queued. The next four 'instance $s()$ consuming and sending' give the performance of a call to $s()$ with the same chords as above, except that the consumed asynchronous messages are immediately sent back, using for instance **void** $s()$ **& async** $a()$ { $a()$; }.

—As one would expect, sending or consuming an empty asynchronous message has a cost comparable to that of calling a conventional synchronized method. (The fact the measured cost for sending a parameterless message is shown as less than that of making a synchronized call illustrates the limits of this sort of benchmarking on modern architectures.)

—Consuming a message with an **int** parameter is about 3 times slower than consuming a 'signal' with no parameter: this represents the cost of the queue operations, unboxing, and garbage collection. Sending an integer message is shown as just

[12]In our experiments, instance methods are a little slower than static methods and a little faster than virtual ones. The differences are not significant, so we omit those figures.

under 10 times the cost of a synchronized call. Much of this cost is attributable to the repeated boxing of the integer so that it may be stored in the queue. The figures for strings, which do not need boxing, are more reasonable.

—Consuming and sending an empty asynchronous message is in some sense the Polyphonic C♯ equivalent of acquiring and releasing an object lock, and is about 2.4 times slower. This is roughly what one would expect, since both the consumption and the sending involve a synchronized call. Consuming and sending *two* empty asynchronous messages costs about 4 times the cost of a single synchronized call: although both messages are consumed under one lock, each of the sends involves acquiring and releasing the lock again.

—Using the Polyphonic C♯ mutual-exclusion idiom of passing the state in an asynchronous message that is consumed at the start of a chord and then resent at the end costs 5.7 (for a single integer) or 5.3 (for a string) times as much as a synchronized call.

The ping pong and bounded buffer benchmarks compare the performance of Polyphonic C♯ programs that perform synchronization between multiple threads. In the ping pong benchmark, two threads are each associated with a semaphore and repeatedly block and wait for one another—alternately signalling the other semaphore and waiting on their own. The polyphonic version implements the semaphores using

```
public class PSem : Sem {
   public async Signal() & public void Wait() {}
}
```

whereas the plain C♯ version uses

```
public class LSem : Sem {
  int i = 0;
  public void Signal() { lock(this) { i++; Monitor.Pulse(this); }}
  public void Wait() {
    lock(this) {
      while (i == 0) { Monitor.Wait(this); }
      i--;
    }
  }
}
```

On a single processor, the polyphonic version of this test has about half the performance of the handwritten, non-polyphonic version. (Using the thread interruption-based implementation of *ThreadQ*, the polyphonic version runs nearly 10 times slower than the non-polyphonic version. On the previous version of .NET, however, this ratio was nearly inverted as the non-polyphonic version ran 85 times slower than it does on the current one.) On a dual processor, the two versions of the program run at roughly the same speed.

In the bounded buffer benchmark, a number of producer threads fill a 100-element bounded buffer, whilst the same number of consumer threads remove those elements. Producers and consumers block as the buffer becomes full and empty, respectively. The polyphonic bounded buffer uses

```
public class PBB : Buffer {
  public void Put(string s) & async free(int c) {
    if (c==1) { full(); } else { free(c-1); }
    p(s);
  }
  public string Get() & async p(string s) & async full() {
    free(1);
    return s;
  }
  public string Get() & async p(string s) & async free(int c) {
    free(c+1);
    return s;
  }
  public PBB(int capacity) { free(capacity); }
}
```

whereas the plain C♯ version uses

```
public class LBB : Buffer {
  private Queue q = new Queue();
  private int capacity;
  public void Put(string s) {
    lock(this) {
      while (q.Count == capacity) { Monitor.Wait(this); }
      q.Enqueue(s);
      Monitor.PulseAll(this);
    }
  }
  public string Get() {
    lock(this) {
      while (q.Count == 0) { Monitor.Wait(this); }
      string s = (string)(q.Dequeue());
      Monitor.PulseAll(this);
    }
    return s;
  }
  public LBB(int capacity) { this.capacity = capacity; }
}
```

On a single-processor machine, the polyphonic solution significantly outperforms the plain solution, with either two or four threads. On a dual-processor machine, the two solutions are roughly equivalent with two threads, and the polyphonic solution is three times faster with four threads. Although these results crucially depend on the underlying scheduling on threads, we interpret the speedup as a consequence of selected wakeups in polyphonic code. Of course, C♯ code optimized by hand would eventually outperform any Polyphonic C♯ code.

6.3 Remarks on Concrete Syntax

There is some redundancy in the concrete syntax of Polyphonic C♯ as presented here: the attributes, modifiers and return type information of each method have to be repeated (consistently) for each chord in which the method appears. One alternative approach (essentially that of Funnel [Odersky 2000]) would be to allow synchronous method definitions to have more than one body, each of which is guarded by a purely asynchronous pattern, and to specify modifiers and attributes of asynchronous methods in separate declarations. In this style, the reader-writer lock of Section 4.2 could look something like this:

```
class ReaderWriter {
  async idle();    // Just signatures. Any modifiers or
  async s(int);    // attributes would occur here too

  ReaderWriter() { idle(); }

  public void Shared()
      when idle() { s(1); }
      when s(int n) { s(n+1); }

  public void ReleaseShared()
      when s(int n) {
          if (n == 1) idle(); else s(n-1);
      }
  public void Exclusive()
      when idle() {}

  public void ReleaseExclusive() { idle(); }
}
```

This alternative syntax is more compact in some cases (e.g. in subclasses of *ActiveObject*), but is also less flexible: one must group chords by synchronous methods (rather than, for example, by asynchronous state) and it is awkward to turn **void** into **async** or vice-versa.

6.4 Future Work

Amongst the areas for further work on Polyphonic C♯ that we think are particularly interesting are:

Concurrency Types. As suggested in our examples, it is relatively easy to state and verify invariants in polyphonic classes, often from the shape of the chords and the visibility of their methods.

Several type systems and other static analyses have been developed in similar settings to automate the process, and check (or even infer) at compile time some behavioural properties such as

(1) There is one, or at most one, pending message for this asynchronous method, or for this set of methods.
(2) Calls to this method are always eventually processed (partial deadlock-freedom).

The potential benefits are obvious: the compiler can catch more programming errors, and otherwise produce more efficient code. While these tools are still rather complex, this is a very active area of research in concurrency [Nielson and Nielson 1994; Igarashi and Kobayashi 2001; Chaki et al. 2002]. Needless to say, it would be much more difficult to check those properties on a code that directly uses threads and locks instead of chords.

Optimizations. There are many opportunities for optimizing the implementation described here. Some of these require proper static analysis, whereas others could usefully be implemented on the basis of more naive compile-time checks:

—Lock optimization. There are situations when we could safely 'fuse' successive critical sections protected by the same lock, for example when several a (bounded) series of asynchronous messages are sent to the same object, or when a chord body immediately sends messages to **this**.

—Queue optimizations. Polyphonic methods for which it can be determined that there can be at most one pending call on a particular object could be compiled using private fields instead of queues. Similarly, the same queue could be shared by mutually-exclusive methods.

—Thread optimization. Purely asynchronous chords that only perform very brief terminating computations (such as sending other messages) can often be compiled to run in the invoking thread, rather than a new one. This is a desirable optimization, since it is not uncommon to have a public method that arguably *should* be asynchronous and which merely synchronizes with, and then sends, other (typically private) asynchronous messages. In such cases, one usually prefers not to pay the cost of thread startup and so defines the method as **void** rather than **async**, although this damages compositionality, for example by preventing one from instantiating an **async** delegate with the method. Concrete examples of this situation are provided by the *ReleaseShared* and *ReleaseExclusive* methods of the *ReaderWriter* class from Section 4.2—although the potentially-blocking calls to *obtain* the lock clearly have to be synchronous, the methods for *relinquishing* it could safely and neatly be made asynchronous were it not for the fact that they would then be handled by an expensive (and transient) new thread. Unfortunately, using static analysis to detect that a non-trivial chord body always terminate 'quickly' is rather hard, so it may be that programmer annotation is a better solution to this problem.

Pattern-Matching. There are situations in which it would be convenient to specify chords that are only enabled if the *values* passed as arguments to the methods satisfy additional constraints. A typical example concerns asynchronous messages with embedded sequence numbers. If one wishes to ensure that such messages are processed in sequence, then one currently has to manage a new queue of unprocessed messages by hand. For example, we may extend *ActiveObject* of Section 4.4 with timestamps as follows:

```
class SequenceProcessor : ActiveObject {
  private Hashtable pending = new Hashtable();
  private int next = 0;
```

```
    public async Message(int stamp, string contents)
    & override protected void ProcessMessage() {
      if (stamp == next) {
        DealWith(contents);
        while (pending.ContainsKey(++next)) {
            DealWith((string)pending[next]);
            pending.Remove(next);
        }
      } else {
        pending.Add(stamp,contents);
      }
    }
    ...
}
```

With pattern-matching, one could achieve the same effect with

```
class SequenceProcessor : ActiveObject {

    public async Message(int stamp, string contents)
    & override protected void ProcessMessage()
    & async waitingfor(int stamp) {
        DealWith(contents);
        waitingfor(stamp++);
    }
    SequenceProcessor() {
      waitingfor(0);
    }
    ...
}
```

in which the *stamp* parameters of the two asynchronous calls are required to be equal for the pattern to match. Even more useful is to allow more general guard constraints to be added to chords, as in the following example, which matches buyers for an item with sellers of the same item, provided the bid price is greater than or equal to the offer price:

```
override protected void ProcessMessage()
& public async Bid(string bidname, int bidprice, int bidvol, Client bidder)
& public async Offer(string offname, int offprice, int offvol, Client seller)
& ((bidname == offname) && (bidprice >= offprice)) {
    // actually do a deal
    int dealvolume = min(bidvol, offvol);
    int dealprice = avg(bidprice, offprice);
    seller.sale(bidname, dealprice, dealvolume);
    bidder.purchase(bidname, dealprice, dealvolume);
    ...
}
```

We have implemented an experimental extension of our compiler that will compile code like that given above. The extended language is elegant and expressive but would need to be compiled more efficiently to be suitable for general-purpose use: matches are found by an expensive sequential search through all combinations of calls to the methods appearing in a chord. Other languages which allow guarded acceptance of messages, such as SR [Andrews et al. 1988; Andrews and Olsson 1993], have used linear traversal of message queues, but since more than one queue is involved in a chord we potentially have a more expensive search.[13]

To make the behaviour of guarded chords predictable and also to facilitate possible optimizations, it seems reasonable to restrict guards to a simple sublanguage, such as conjunctions of equalities and inequalities over a few primitive types, rather than allow them to be arbitrary boolean expressions. Guards should evaluate quickly and be free of side-effects (reading or writing mutable state, throwing exceptions, blocking and so on). Efficient incremental matching of conjunctive queries (or constraints) over primitive scalar types has been previously studied in the contexts of databases, constraint logic programming languages and rule-based systems, though the existing algorithms of which we are aware are not immediately suitable for use in a setting in which the values contributing to a match are immediately deleted. This seems a promising area for future work.

Timeouts and Priorities. Similarly, it is tempting to supplement the syntax for chords with some declarative support for priorities or timeouts and, more generally, to provide a finer control over dynamic scheduling. We are considering several designs for timeouts (essentially generalizing the notion of guard discussed above) and are investigating implementation trade-offs.

6.5 Related Work

There is a large literature on programming abstractions and language constructs for concurrency. We do not attempt a comprehensive survey—see for instance [Philippsen 1995] for concurrent object-oriented languages. Instead, we first discuss works closely related to our design, then we compare it to some popular approaches in concurrent object-oriented programming.

Banâtre et al. [1988] originally proposed to program concurrent and distributed systems by multiset transformation of messages in a 'chemical soup', using 'reaction rules' defined by pattern matching on pending messages. These reaction rules can be seen as early, top-level ancestors for 'chords' with handwritten, architecture-dependent implementations.

Our work is directly inspired by the join calculus, a formalism with strong connections to concurrency theory and functional programming [Fournet and Gonthier 1996]. The main purpose of the join calculus is to define a core calculus for asynchronous programming, as opposed to more abstract specification languages for concurrency. In the same spirit, Pierce and Turner [2000] developed a programming language entirely based on communications on pi calculus channels. The join calculus has been used to design concurrent, distributed, and mobile extensions of

[13]It should be noted, however, that a language which only allows guarded acceptance of single messages but allows those guards to refer to values which can be mutated may have to repeatedly search whenever the values of those variables change.

functional programming languages [Le Fessant and Maranget 1998; Conchon and Le Fessant 1999]. More abstractly, Buscemi and Sassone [2001] precisely related several standard classes of Petri nets to typed subsets of the join calculus, essentially showing that join patterns (or chords here) can express dynamically-evolving Petri nets.

The work that is most closely related to Polyphonic C♯ is that on Join Java [Itzstein and Kearney 2001; Itzstein and Kearney 2002]. Join Java, which was initially designed at about the same time as Polyphonic C♯, takes almost exactly the same approach to integrating join calculus in a modern object-oriented language. Apart from minor variations of syntax, the main language differences appear to be that Join Java takes a more restrictive approach to inheritance than Polyphonic C♯ (simply outlawing inheritance from any class that uses join patterns) and that Join Java also allows the programmer to specify whether pattern matching within a class should be sequential or non-deterministic. The implementation of Join Java uses a tree-based pattern-matching library; some further details are given by Itzstein and Jasiunas [2003].

Going back to Simula [Dahl and Nygaard 1966], concurrent message-passing was one of the original interpretations of objects. However, mainstream object-oriented languages tend to focus on lower-level mechanisms such as shared-memory, locks, and threads.

In order to facilitate the use of threads and locks, most programming languages provide support for allocating locks on demand (such as synchronized objects), acquiring and releasing locks consistently (such as synchronized methods), and selectively acquiring locks (such as monitors or conditional critical regions). In contrast, we provide general abstractions for synchronizing and atomically consuming several messages sent on distinct methods of the same (dynamically-allocated) object—typically from independent threads—and we hide the usage of locks in the implementation.

Many popular synchronization patterns embedded in existing programming languages can be concisely expressed as chords. For instance, we provided examples of *synchronization barriers*, also found in dataflow languages and modelled in Petri nets (Section 4.3), of *active objects*, also found in Actor languages [Hewitt 1977] (Section 4.4), and of staged computations (Section 4.5). Similarly, we can also encode ADA-style rendezvous [Fournet and Gonthier 2002] and Linda-style coordination primitives using chords.

Several languages for concurrency provide expressive guards that can be used to control the acquisition of exclusive resources and test local conditions while entering critical regions [Agha et al. 1993; Andrews and Olsson 1993]. Polyphonic C♯ explores a different approach, and emphasizes the synchronization of multiple messages. However, programming synchronization in terms of boolean conditions is often natural—indeed, we are experimenting with such extensions. Using guards *instead* of chords, one could in principle multiplex all calls to polyphonic methods in a given object to methods whose guards express equivalent synchronization conditions. However, this would involve complicated guards, difficult to write correctly and compile efficiently.

Other languages provide generic support for creating and composing message handlers. For example, [Reppy 1992] provides ML libraries that support the com-

positional, higher order definition of synchronous event channels. Although it is possible to program event filters similar to chords, their implementation is also less specialized and less efficient, and would typically involve an explicit representation of message queues.

7. CONCLUSIONS

Asynchronous concurrent programming is becoming more important and widespread but remains hard. We have designed and implemented a join calculus-based extension of C♯ that is simple, expressive, and efficient. In our experience, writing correct concurrent programs is considerably less difficult in Polyphonic C♯ than in ordinary C♯ (though we would certainly not go so far as to claim that it is easy!).

The integration of the join calculus constructs with objects and the existing .NET mechanisms for concurrency is not entirely straightforward—our implementation is constrained by the underlying threads-and-locks model. Some uses of polyphony with existing libraries also require a little 'impedance matching'. Nevertheless, the new constructs work very well in practice.

ACKNOWLEDGMENT

Thanks to Mark Shinwell, who implemented the first prototype of the Polyphonic C♯ compiler during an internship at Microsoft Research in 2000, to Claudio Russo for his work on the current implementation, to Hernan Melgratti for prototyping guarded matching, and to the anonymous referees.

REFERENCES

Agha, G. 1990. *ACTORS : A model of Concurrent computations in Distributed Systems.* The MIT Press, Cambridge, Mass.

Agha, G., Wegner, P., and Yonezawa, A. 1993. *Research Directions in Concurrent Object-Oriented Programming.* MIT Press.

America, P. 1989. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing 1,* 4, 366–411.

Andrews, G. R. and Olsson, R. A. 1993. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings.

Andrews, G. R., Olsson, R. A., Coffin, M., Elshoff, I., Nilsen, K., Purdin, T., and Townsend, G. 1988. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems 10,* 1 (Jan.), 51–86.

Banâtre, J.-P., Coutant, A., and Métayer, D. L. 1988. A parallel machine for multiset transformation an its programming style. *Future Generation Computing Systems 4*, 133–144.

Barclay, T., Gray, J., and Slutz, D. 2000. Microsoft TerraServer: A spatial data warehouse. In *Proceedings of ACM SIGMOD.* Also Microsoft Research Tech Report MS-TR-99-29.

Barclay, T., Gray, J., Strand, E., Ekblad, S., and Richter, J. 2002. TerraService.NET: An introduction to web services. Tech. Rep. MSR-TR-2002-53, Microsoft Research. June.

Ben-Ari, M. 1998. How to solve the Santa Claus problem. *Concurrency: Practice & Experience 10,* 6, 485–496.

Benton, N. 2003. Jingle bells: Solving the Santa Claus problem in Polyphonic C♯.

Birrell, A. D. 1989. An introduction to programming with threads. Research Report 35, DEC SRC. Jan.

Birrell, A. D., Guttag, J. V., Horning, J. J., and Levin, R. 1987. Synchronization primitives for a multiprocessor: A formal specification. Research Report 20, DEC SRC. Aug.

Buscemi, M. G. and Sassone, V. 2001. High-level petri nets as type theories in the join calculus. In *Foundations of Software Science and Computation Structures, 4th International Conference*

(FOSSACS 2001), F. Honsell and M. Miculan, Eds. Lecture Notes in Computer Science, vol. 2030. Springer, 104–120.

Cardelli, L. and Davies, R. 1999. Service combinators for web computing. *Software Engineering 25,* 3, 309–316.

Chaki, S., Rajamani, S. K., and Rehof, J. 2002. Types as models: Model checking message-passing programs. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM.

Conchon, S. and Le Fessant, F. 1999. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*. IEEE Computer Society, 22–29. Software and documentation available from http://pauillac.inria.fr/jocaml.

Dahl, O.-J. and Nygaard, K. 1966. SIMULA – an ALGOL-Based Simulation Language. *Communications of the ACM 9,* 9 (Sept.), 671–678.

Detlefs, D. L., Leino, K. R. M., Nelson, G., and Saxe, J. B. 1998. Extended static checking. Research Report 159, DEC SRC. Dec.

ECMA. 2001. Standard ECMA-334: C♯ Language Specification.

Fournet, C. and Gonthier, G. 1996. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM-SIGACT Symposium on Principles of Programming Languages*. ACM, 372–385.

Fournet, C. and Gonthier, G. 2002. The join calculus: a language for distributed mobile programming. In *Proceedings of the Applied Semantics Summer School (APPSEM), Caminha, Sept. 2000*, G. Barthe, P. .Dybjer, , L. Pinto, and J. Saraiva, Eds. Lecture Notes in Computer Science, vol. 2395. Springer-Verlag, 268–332.

Fournet, C., Laneve, C., Maranget, L., and Rémy, D. 2000. Inheritance in the join-calculus (extended abstract). In *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*. Lecture Notes in Computer Science, vol. 1974. Springer-Verlag, 397–408. Full version to appear in Journal of Logic and Algebraic Programming.

Gosling, J., Joy, B., and Steele, G. 1996. Threads and locks. In *The Java Language Specification*. Addison Wesley, Chapter 17.

Gurevich, Y., Schulte, W., and Wallace, C. 2000. Investigating Java concurrency using abstract state machines. In *Abstract State Machines: Theory and Applications*, Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, Eds. Lecture Notes in Computer Science, vol. 1912. Springer-Verlag, 151–176.

Hewitt, C. 1977. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence 8,* 3, 323–364.

Hoare, C. A. R. 1974. Monitors: An operating system structuring concept. *Communications of the ACM 17,* 10 (Oct.), 549–557.

Hoare, C. A. R. 1985. *Communicating Sequential Processes*. Prentice-Hall.

Igarashi, A. and Kobayashi, N. 2001. A generic type system for the Pi-Calculus. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM.

INMOS Limited. 1984. *Occam Programming Manual*. Prentice-Hall Int.

Itzstein, G. S. and Jasiunas, M. 2003. On implementing high level concurrency in java. In *Proceedings of the Eighth Asia-Pacific Computer Systems Architecture Conference*. Japan. To appear.

Itzstein, G. S. and Kearney, D. 2001. Join Java: An alternative concurrency semantics for Java. Tech. Rep. ACRC-01-001, University of South Australia.

Itzstein, G. S. and Kearney, D. 2002. Applications of Join Java. In *Proceedings of the Seventh Asia-Pacific Computer Systems Architectures Conference (ACSAC2002), Melbourne, Australia*, F. Lai and J. J. Morris, Eds. Conferences in Research and Practice in Information Technology, vol. 6. ACS, 37–46.

Kamin, S., Ed. 1997. *Proceedings of the First ACM-SIGPLAN Workshop on Domain-Specific Languages*. Paris, France.

Larus, J. R. and Parkes, M. 2001. Using cohort scheduling to enhance server performance. Tech. Rep. MSR-TR-2001-39, Microsoft Research. Mar.

Le Fessant, F. and Maranget, L. 1998. Compiling join-patterns. In *HLCL '98: High-Level Concurrent Languages*, U. Nestmann and B. C. Pierce, Eds. Electronic Notes in Theoretical Computer Science, vol. 16(3). Elsevier Science Publishers.

Lea, D. 1999. *Concurrent Programming in Java: Design Principles and Patterns*, Second Edition ed. Addison-Wesley.

Matsuoka, S. and Yonezawa, A. 1993. Analysis of inheritance anomaly in object-oriented concurrent programming languages. See Agha et al. [1993], Chapter 4, 107–150.

Nielson, H. R. and Nielson, F. 1994. Higher-order concurrent programs with finite communication topology. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM.

Odersky, M. 2000. Functional nets. In *Proceedings of the European Symposium on Programming*. Lecture Notes in Computer Science, vol. 1782. Springer-Verlag, 1–25.

Philippsen, M. 1995. Imperative concurrent object-oriented languages: An annotated bibliography. Tech. Rep. TR-95-049, International Computer Science Institute, Berkeley, CA.

Pierce, B. C. and Turner, D. N. 2000. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. D. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press.

Ramming, J. C., Ed. 1997. *Proceedings of the First USENIX Conference on Domain-Specific Languages*. Santa Barbara, California.

Reppy, J. H. 1992. Concurrent ML: Design, application and semantics. In *Programming, Concurrency, Simulation and Automated Reasoning*. Lecture Notes in Computer Science, vol. 693. Springer-Verlag, 165–198.

Trono, J. A. 1994. A new exercise in concurrency. *SIGCSE Bulletin 26,* 3, 8–10. Corrigendum: 26(4):63.

Yonezawa, A. 1990. *ABCL: An Object-Oriented Concurrent System – Theory, Language, Programming, Implementation and Application*. Computer System Series. MIT Press.