Join Patterns for Visual Basic

Claudio V. Russo
Microsoft Research Ltd., Cambridge, U.K.
crusso@microsoft.com

Abstract

We describe an extension of Visual Basic 9.0 with asynchronous concurrency constructs - join patterns - based on the join calculus. Our design of Concurrent Basic (CB) builds on earlier work on Polyphonic C$^\#$ and Cω. Since that work, the need for language-integrated concurrency has only grown, both due to the arrival of commodity, multi-core hardware, and the trend for Rich Internet Applications that rely on asynchronous client-server communication to hide latency. Unlike its predecessors, CB adopts an event-like syntax that should be familiar to existing VB programmers. Coupled with Generics, CB allows one to declare re-useable concurrency abstractions that were clumsy to express previously. CB removes its ancestors' inconvenient inheritance restriction, while providing new extensibility points useful in practical applications that must co-exist with or want to exploit alternative threading models available on the platform. CB is implemented as an extension of the production VB 9.0 compiler.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language constructs and features—Concurrent programming structures;control structures; classes and objects

General Terms Languages

Keywords Visual Basic, join patterns, asynchronous message passing.

1. Introduction

This paper presents Concurrent Basic (CB), an extension of Visual Basic 9.0 (11) with asynchronous concurrency constructs - join patterns - derived from the join calculus (5). The name CB is a deliberate pun on the channel-based Citizen's Band radio popular in Basic's heyday. Our motivation is to let VB rise to the occasion of both parallel programming on multi-core hardware, and distributed programming of Rich Internet Applications, using asynchronous communication to hide latency. Unlike its C$^\#$ based predecessors, Polyphonic C$^\#$ (4) and Cω (12), CB adopts an accessible, event-like syntax that should be familiar to modern VB programmers. By exploiting Generics in Visual Basic, CB allows one to declare parametric, thus reusable, concurrency abstractions that were awkward to express, less efficient and less safe in the original C$^\#$ proposals: these pre-dated Generics and had to resort to expensive boxing and failure-prone casts. CB offers more natural support for inheritance, enabling a subclass to augment the set of patterns declared on inherited channels: the C$^\#$ variants forced users to copy existing patterns into the subclass, breaking encapsulation. CB also provides novel, semi-declarative hooks to let users control the execution of patterns as an ad-hoc, yet important, concession to manual optimization and the integration with other concurrency frameworks available on the platform. The CB compiler is an extension of the production Visual Basic 9.0 compiler that generates code for the Common Language Runtime (CLR) with performance similar to C$^\#$. Our compiler is available both from the command-line and from within Visual Studio.

Although less fashionable amongst researchers, Visual Basic is hugely popular. Having evolved into a statically typed, class-based object-oriented language there is very little to distinguish VB 9.0 from C$^\#$ 3.0. Both support Generics, type inference, language integrated query (LINQ), and lambda expressions. Where they currently differ is in C$^\#$'s support for CLU-style iterators, unsafe code, and lambda *statements*. VB, for its part, provides optional and named argument passing, Cω-style XML literals, dynamic typing and late-binding (runtime member resolution).

Interestingly, one of the features that has distinguished VB and C$^\#$ from the start is VB's support for declarative event handling. Events support a simple publish/subscribe model: a class publishes some internally raised *events* that zero or more, typically unknown, subscribers can *handle*. While both languages let you define event members, only VB allows a method to *declare* that it *handles* some named events - C$^\#$ users have to add and remove event callbacks imperatively.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

```
Public Class Document
  Inherits Form
  Private WithEvents SaveButton As Button
  Private WithEvents SubmitButton As Button
  Public Event Saved( sender As Object, args As EventArgs)
  Private Sub CaseSaveOrSubmit( sender As Object, args As EventArgs) _
      Handles SaveButton.Click, SubmitButton.Click
    ' Save this document to disk
    RaiseEvent Saved(Me, Nothing)
  End Sub
  Public Event Submitted( sender As Object, args As EventArgs)
  Private Sub CaseSubmit( sender As Object, args As EventArgs) Handles SubmitButton.Click
    ' Submit this document to the server
    RaiseEvent Submitted(Me, Nothing)
  End Sub
End Class
```

Figure 1. Declarative event handling in Visual Basic.

For instance, a VB GUI application might declare the class `Document` in Figure 1.[1] This class publishes two 2-argument events, `Saved(s,e)` and `Submitted(s,e)`, raised by methods `CaseSaveOrSubmit` and `CaseSubmit`. Method `CaseSaveOrSubmit` is a *handler* that runs whenever the button in variable `SaveButton` or `SubmitButton` raises its `Click` event.[2] This relationship is established by the `Handles` keyword after the method header, followed by a list of qualified event identifiers. The `Click` events (not shown) have the same argument signature as the handler. A single event may have more than one handler. When the event is raised, all of its handlers are executed, on the raising thread and in the order in which they were added to the event. Here, `CaseSaveOrSubmit` and `CaseSubmit` will both run (sequentially) in response to `SubmitButton.Click`. Since events may have zero or many registered handlers they are, by design, not allowed to have a return type.[3]

CB leverages the Basic programmer's familiarity with event-based programming but then turns it on its head. In addition to events, a CB class can declare *channels* on which to *receive* asynchronous messages and synchronous requests. A method may be declared to execute *when* communication has occurred on a particular set of local channels, forming a *join pattern*. Communications are queued until or unless some method is enabled and run. More precisely, a message that enables some join pattern causes its method to run (on some other thread) otherwise the message is queued until one of its patterns is enabled; a request that enables some join pattern runs its method (on the same thread) otherwise the request blocks until one of its patterns is enabled. Unlike an event handler, which services one of several alternative events at a time, in conjunction with all other handlers on that event, a join pattern waits for a conjunction of channels and competes for execution with any other enabled pattern.

Figure 2 is the simplest interesting example of CB, a module declaring a thread-safe, unbounded, unordered string buffer. This example presents all three keywords introduced by CB: `Asynchronous`, `Synchronous` and `When` (which already exists for exception handling). This module declares two channels: an asynchronous channel, `Put(s)`, which takes a string argument and (like all asynchronous methods) returns no result; and a synchronous channel, `Take()`, which takes no arguments but returns a string. The private method `CaseTakeAndPut` is the continuation of a join pattern that can run only `When` both the `Take` and `Put` methods have been called. (The name `CaseTakeAndPut` is arbitrary; as a convention, we choose to derive the name of the continuation from the channels following `When`.)

Suppose several producer and consumer threads wish to communicate via the `Buffer` module. Producers call `Buffer.Put(s)` to post a string, without blocking. Consumers call `Buffer.Take()` to request a string, possibly blocking. Once `Buffer` has received both a `Put(s)` and a `Take()` the body of `CaseTakeAndPut` can run, returning the actual argument `s` of `Put` as the result of the call to `Take`. Since this is the only applicable pattern, a caller of `Take` will wait until or unless a call to `Put` has arrived. Multiple calls to `Take` may be pending before a `Put` is received to reawaken one of them, and multiple calls to `Put` may be queued before their arguments are consumed by a subsequent `Take`. Note that:

1. The body of the continuation runs in the (reawakened) thread corresponding to the matched call to `Take`. Hence no new threads are spawned in this example.

[1] VB is line-based, `_` is the line-continuation marker.

[2] The `WithEvent` modifiers on the button declarations expose their events for event handling. Since `WithEvent` variables are mutable, setting the value of a `WithEvents` variable implicitly causes its handlers to be removed from the current value's event and added to the new value's event.

[3] VB includes a few more event-related constructs: property-like *custom* event declarations and imperative `AddHandler` and `RemoveHandler` statements. We can ignore these for our purposes.

```
Module Buffer
  Public Asynchronous Put( s As String)
  Public Synchronous Take() As String
  Private Function CaseTakeAndPut( s As String) As String When Take, Put
    Return s
  End Function
End Module
```

Figure 2. An unbounded, unordered string buffer in CB.

2. The code produced for the `Buffer` module is completely thread-safe. The compiler generates the necessary locking to ensure that channel invocations are consumed atomically. Furthermore, this locking is fine-grained and brief - method `CaseTakeAndPut` does not lock the entire `Buffer` module during its execution. More generally, continuation methods are not executed with "monitor semantics", acquiring and releasing their object's lock like Java's synchronized methods.
3. The result of a continuation is returned on its synchronous channel, of which there can be at most one following `When`.

In general, a channel may be involved in more than one join pattern, each of which defines a different continuation that may run when the channel is invoked (provided the rest of the pattern is enabled). For example, in the `Either` module of Figure 3, calls to `Receive` synchronize with calls to `First` or `Second`. Now we have two asynchronous channels and a synchronous channel that can wait for either one, with a different continuation and argument type in each case.

A single join pattern may involve more than one asynchronous channel; module `Both` in Figure 4 contains one synchronous join pattern that waits for messages on both `First` and `Second` (as well as `Receive`). Notice that it takes two parameters, one from each channel.

A join pattern may also be purely asynchronous, provided its continuation is a subroutine and its `When` clause only lists asynchronous channels. The pattern `CaseAsyncTakeAndPut` in Figure 5 spawns a new thread that executes the callback `c(i)` whenever messages arrive on both `AsyncTake` and `Put` (in any order).[4] Merging the declarations of `Buffer` and `AsyncBuffer` would yield a module that supports both synchronous and asynchronous consumers.

The paper is structured as follows: Section 2 describes our proposal in more detail; Section 3 gives more motivating examples; Section 4 describes our implementation; Section 5 considers two pragmatic extensions (custom dispatch of patterns and synchronous rendezvous); Section 6 surveys related work and concludes.

[4] The `Delegate` declaration defines a nominal type, `Callback`, for first-class methods matching the specified signature. Delegate values are constructed by applying VB's `AddressOf` operator to a named static or instance method or, in VB 9.0, by writing an anonymous lambda-expression.

2. Detailed Proposal

CB extends VB with just two new constructs: channels and declarative message handling. The syntactic extensions to VB 9.0 (11) appear in Figure 6.

Channels are used to synchronize and pass messages between concurrently executing threads. A channel declaration consists of any optional attributes and access modifiers, optional `Shadows` or `Shared` keyword, followed by keyword `Asynchronous` or `Synchronous`, a method signature (identifier, parameter list and optional return type) and an optional `Implements` clause. If the channel is `Asynchronous`, it must not have a return type. A channel may not declare type parameters. The parameter list may not contain `ByRef`, `Optional` or `ParamArray` parameters. A channel may not overload another member of the same name. Channels may be declared in modules, classes and structs. A channel in a struct must be marked `Shared`. A channel may implement an interface method of a matching signature, but there is no special syntactic support for specifying channels in interfaces.

From a client's perspective, a channel just declares a method of the same name and signature. The client posts a message or issues a request by invoking the channel as a method.

Methods can declaratively service requests and consume messages arriving on channels belonging to the same instance or type. To do so, a method declaration specifies the `When` keyword and lists one or more (distinct) channels. A `When` clause may appear wherever a `Handles` or `Implements` clause can in the existing VB syntax. A channel in the `When` list is specified by an identifier, possibly prefixed by `MyClass`, `MyBase` or `Me`. The (qualified) identifier must denote a `Synchronous` or `Asynchronous` channel declared in the containing type of the method or one of its base classes. The `When` clause of a `Shared` method may only mention `Shared` channels, conversely, the `When` clause of a non-`Shared` method may only mention non-`Shared` channels. The argument types of the method must exactly match the concatenation of the (inherited) argument types of the channels in the `When` clause. The return type of the method must match the return type of the first channel following `When`. Only the first channel following a `When` clause may be `Synchronous`, all subsequent channels must be `Asynchronous`. A type inherits all non-`Shared` `When` clauses provided by its base type.

```
Module Either
  Public Asynchronous First( s As String)
  Public Asynchronous Second( i As Integer)
  Public Synchronous Receive() As String
  Private Function CaseReceiveAndFirst( s As String) As String When Receive, First
    Return s
  End Function
  Private Function CaseReceiveAndSecond( i As Integer) As String When Receive, Second
    Return i.ToString()
  End Function
End Module
```

Figure 3. Waiting on a disjunction of channels.

```
Module Both
  Public Asynchronous First( s As String)
  Public Asynchronous Second( i As Integer)
  Public Synchronous Receive() As String
  Private Function CaseReceiveAndFirstAndSecond( s As String, i As Integer) As String _
      When Receive, First, Second
    Return s + i.ToString()
  End Function
End Module
```

Figure 4. Waiting on a conjunction of channels.

```
Module AsyncBuffer
  Public Delegate Sub Callback( i As Integer)
  Public Asynchronous AsyncTake( c As Callback)
  Public Asynchronous Put( i As Integer)
  Private Sub CaseAsyncTakeAndPut( c As Callback, i As Integer) When AsyncTake, Put
    c(i)
  End Sub
End Module
```

Figure 5. Spawning new tasks.

A derived type cannot in any way alter the `When` clauses it inherits from its base type, but may declare additional ones.

Intuitively, a continuation method must wait until/unless a single request or message has arrived on each of the channels following the continuation's `When` clause. If the continuation gets to run, the arguments of each channel invocation are dequeued (thus consumed) and transferred (atomically) to the continuation's parameters. For this reason, the method's argument signature must be compatible with the concatenation of the argument signature of its channels. Similarly, when the continuation returns, it will return its value to the invoker of the synchronous channel (if any). Thus the return type of the continuation must match the return type of any synchronous channel in its `When` clause.

A type may declare multiple channels and multiple join patterns on subsets of those channels. Patterns may (but are not recommended to) specify overlapping sets of channels, in which case the patterns will compete for execution. A subclass may extend its set of inherited channels and/or reference accessible channels of its base classes in its own patterns.

The act of invoking a channel may enable zero, one or more join patterns involving that channel:

- If no pattern is enabled then the channel invocation is queued up. If the channel is asynchronous, then this simply involves adding the arguments of the invocation (the contents of the message) to a queue. If the channel is synchronous, then the calling thread is blocked, joining a notional queue of threads waiting on this channel.
- If there is a single enabled pattern, then the arguments of the channels involved in the match are de-queued, and any blocked thread involved in the match is awakened to run the join pattern's continuation in that thread. The continuation of a join pattern involving only asynchronous channels is run in a newly spawned thread.

ModuleMemberDeclaration	:=	... | *ChannelMemberDeclaration*
ClassMemberDeclaration	:=	... | *ChannelMemberDeclaration*
StructMemberDeclaration	:=	... | *ChannelMemberDeclaration*
*ChannelMemberDeclaration**	:=	[*Attributes*] [*ChannelModifiers*+] *ChannelSignature* [*ImplementsClause*] [*StatementTerminator*]
*ChannelSignature**	:=	Asynchronous *Identifier* [(*[ParameterList]*)]
	|	Synchronous *Identifier* [([*ParameterList*])] [*As* [*Attributes*] *TypeName*]
*ChannelModifiers**	:=	*AccessModifier* | Shadows | Shared
HandlesOrImplements	:=	... | *WhenClause*
*WhenClause**	:=	When *WhenList*
*WhenList**	:=	*ChannelMemberSpecifier*
	|	*WhenList*, *ChannelMemberSpecifier*
*ChannelMemberSpecifier**	:=	*Identifier*
	|	*MyBase.Identifier*
	|	*Me.Identifier*
	|	*MyClass.Identifier*

Figure 6. Required extensions to the VB 9.0 syntax (11); nonterminals marked with * are additions.

- If several join patterns are enabled, an unspecified one is selected to run.
- If multiple calls to one channel are queued up, which call will be de-queued by a match is unspecified.

The underspecification in this idealized description is deliberate. While CB's implementation of asynchronous message queues have FIFO semantics, the programmer is not meant to exploit this. In a distributed setting, asynchronous calls issued in one sequential order may well arrive in a different one (due to latency and routing effects). The FIFO order of synchronous calls (from separate threads) is not guaranteed because the underlying CLR/Windows locks used to implement synchronous call queues do not, for performance reasons, guarantee deterministic ordering. Finally, in the implementation, the message that awakens a synchronous call may not be the one consumed by it since the message is actually left available in the queue to be stolen by some intervening thread; indeed, the awakened thread may have to wait again if none of its patterns are enabled by the time it actually wakes up (4).

3. Examples

3.1 Example: A One-Place Buffer

The original `Buffer` module is unbounded: any number of calls to `Put` could be queued up before matching a `Take`. We now describe a variant in which only a single data value may be held in the buffer at any one time. This time, we define a *generic class*, `OnePlaceBuffer(Of T)`, to support multiple instances of varying content types `T` (Figure 7): The public interface of `OnePlaceBuffer(Of T)` is similar to that of `Buffer`, but calling `Put(t)` is now synchronous and will block if the buffer is not empty. The implementation of `OnePlaceBuffer` makes use of two private asynchronous messages: `Empty` and `Contains(t)`. These are used to carry the state of the buffer and illustrate a very common programming pattern in CB. The class is best understood by reading its code declaratively. When a `New` buffer is created, it is initially `Empty()`. If someone calls `Put(t)` on an `Empty()` buffer then it subsequently `Contains(t)` and the call to `Put(t)` returns. If someone calls `Take()` on a buffer which `Contains(t)` then the buffer is subsequently `Empty()` and `t` is returned to the caller of `Take()`. Implicitly, in all other cases, calls to `Put(t)` and `Take()` block. The constructor establishes and the patterns maintain the invariant that there is always exactly one `Empty()` or `Contains(t)` message pending on the buffer.

3.2 Example: Futures

Futures are an established concurrency abstraction used to represent the eventual value of a concurrent computation. Generic futures with explicit waiting are simple to code up with CB (Figure 8). Creating a new `Future(Of T)` from a `Computation` returning a value of type `T` (expressed as a delegate or VB 9.0 lambda expression) spawns a new thread to `Execute` the computation in parallel. When the current (or another) thread actually needs the value of the computation it calls a method on the future to `Wait` until/unless

```
Public Class OnePlaceBuffer(Of T)
  Private Asynchronous Empty()
  Private Asynchronous Contains( t As T)
  Public Synchronous Put( t As T)
  Public Synchronous Take() As T
  Private Sub CasePutAndEmpty( t As T) When Put, Empty
    Contains(t)
  End Sub
  Private Function CaseTakeAndContains( t As T) As T When Take, Contains
    Empty()
    Return t
  End Function
  Public Sub New()
    Empty()
  End Sub
End Class
```

Figure 7. A generic, one-place buffer.

```
Public Class Future(Of T)
  Public Delegate Function Computation() As T
  Public Synchronous Wait() As T
  Private Asynchronous Execute(Comp As Computation)
  Private Asynchronous Done(t As T)
  Private Function CaseWaitAndDone(t As T) As T When Wait, Done
    Done(t)
    Return t
  End Function
  Private Sub CaseExecute(Comp As Computation) When Execute
    Done(Comp())
  End Sub
  Public Sub New(Comp As Computation)
    If Comp Is Nothing Then Throw New ArgumentNullException()
    Execute(Comp)
  End Sub
End Class
```

Figure 8. Generic Futures.

the worker has finished the computation and issued `Done(t)`. Between creating the future and obtaining its value, the current thread is free to perform other tasks. Notice the use of the asynchronous pattern, `CaseExecute`, to spawn a new thread. The motivation for reposting the consumed `Done(t)` message in `CaseWaitAndDone` is to allow multiple `Waits` (i.e. reads) to succeed. Modifying this class to propagate exceptions thrown by `Comp`, cancellation of `Comp` or to execute `Comp` using a thread from the thread pool is straightforward. A useful optimization is to cache the value of `Comp()` in a private field of type `T` protected by an argument-less `Done()` channel. This would be an improvement because an argument-less asynchronous channel is actually represented more efficiently as a count of pending invocations rather than a heap-allocated queue of arguments.

3.3 Example: Active Objects

Active object or actors are a popular pattern for asynchronous programming (1). Active objects communicate by asynchronous messages that are processed sequentially by object-specific threads: each active object runs a private event loop. One way of programming active objects in CB is by inheritance from a common base class, the class `ActiveObject` in Figure 9. Sending a `Start()` message spawns a new thread that loops calling the synchronous channel `ProcessMessage()` waiting until or unless the next enabled pattern on `ProcessMessage` can run. The `Halt` message terminates the loop.

Concrete subclasses of `ActiveObject` declare additional patterns on the protected `ProcessMessage` channel, joining it with locally declared asynchronous messages particular to that class.

```
Public Class ActiveObject
  Private Done As Boolean
  Protected Synchronous ProcessMessage()
  Public Asynchronous Start()
  Private Sub CaseStart() When Start
    While Not Done
      ProcessMessage()
    End While
  End Sub
  Public Asynchronous Halt()
  Private Sub CaseHalt() When ProcessMessage, Halt
    Done = True
  End Sub
End Class
```

Figure 9. The base class for an active object.

To illustrate, here is an extract from a lift (elevator) simulation sample written in CB. The sample is derived from the Erlang Lift controller example (1), but we added some animation and people to drive the simulation. In the code, a subclass of `ActiveObject` is used to represent each type of agent (lift, cabin, floor or person) involved in the simulation.

Figure 10 contains an extract from the `Person` class. It publishes a `GotoFloor()` message whose executions, by synchronizing with `ProcessMessage()`, are guaranteed to be serialized. Hence there is no need to protect the private `floor` (and indeed `Done`) field with a lock.

Notice the use of inheritance: `Person` declares an additional pattern on the inherited `ProcessMessage()` channel. Similarly, a derived class of `Person` will inherit this pattern but is free to declare additional channels and patterns. CB's support for inheritance and incremental extension of patterns is an important feature that distinguishes it from its predecessors Polyphonic C$^{\#}$ and Cω (see Section 6).

Although active objects typically communicate asynchronously, if needed, an object can synchronize with another (here `floor`) by posting, with its message, an explicit acknowledgement, `a`, to wait on (`a.Receive()` waits for an asynchronous call to `a.Send()`). The join based implementation of class `Ack` is trivial and similar to our `Buffer` module. We will revisit and simplify this code in Section 5.2.

Threads are expensive on the CLR; having more than a few hundred *ActiveObjects* around at once will typically exhaust the machine's resources or bring execution to a crawl. However, active objects are not built-in to CB: they are just an example of an abstraction built with join patterns. Implementing variants of this abstraction, such as families of active objects that share a thread pool for better scalability, is also possible.

3.4 Example: Parallel Life

Another application we developed is a simple parallel implementation of Conway's Game of Life. The virtual grid of cells is partitioned, vertically and horizontally, amongst $p \times q$ concurrent nodes. Each node computes an $m \times n$ subregion of the grid using a dedicated worker thread. A node maintains a (double buffered) private array with $(m+2)*(n+2)$ cells, overlapping one edge with each neighbour. To synchronize, a node repeatedly:

- posts its 4 interior edges to its neighbours;
- waits to receive 4 exterior edges on 4 separate channels from its neighbours;
- computes new cell values in parallel with other nodes.

To simplify the algorithm, exterior nodes post exterior edges to themselves.

Figure 11 outlines the code for a `Node` (ignoring boundary conditions). Nodes `Up` through `Left` reference neighbouring nodes. The arrays `TopBuff` through `LeftBuff` are buffers, allocated (just once) in `New()`. The buffers migrate between nodes, but are only owned and modified by one node at a time. `Send` copies the node's computed interior edge values into the buffers, before posting them on its neighbours' channels. `Receive()` waits to receive exterior edge values and then copies them into the current subgrid. `Relax()` computes the next generation of the node's subgrid (code omitted).

Some observations are in order. Since no subgrids are shared, this algorithm is easy to distribute across machines. All synchronization is local: for this reason, two nodes (x_1, y_1) and (x_2, y_2) may be working on different iterations in parallel, but will never be more than $|x_1 - x_2| + |y_1 - y_2|$ iterations apart. Waiting simultaneously on four separate channels is both simpler and more efficient than waiting for one edge to arrive from each neighbour over a single, shared channel. The latter design would require a node to wakeup more often and manually queue edges arriving from different neighbours until it had received at least one edge from each. Finally, `Cell` is a type parameter of an enclosing class, so we can reuse this algorithm for different cell types (i.e. automata).

```
Public Class Person
  Inherits ActiveObject
  Private floor As Floor
  Public Asynchronous GotoFloor(f As Floor)
  Private Sub CaseProcessMessageAndGotoFloor(f As Floor) When ProcessMessage, GotoFloor
    floor = f
    Dim a As New Ack()
    floor.PersonArrived(Me, a)
    a.Receive()
    ChooseDir(floor)
  End Sub
  ...
End Class
```

Figure 10. A derived active object class representing a person in the lift simulation.

```
Class GenericPCA(Of Cell)
  Class Node
    ...
    Private Asynchronous StartWorker()
    Private Sub CaseStartWorker() When StartWorker
      While True
        Send()
        Receive()
        Relax() ' Relax() computes the next subgrid
      End While
    End Sub
    Private TopBuff, RightBuff, BottomBuff, LeftBuff As Cell()
    Public Up, Right, Down, Left As Node
    Public Asynchronous TopChan(Edge As Cell())
    Public Asynchronous RightChan(Edge As Cell())
    Public Asynchronous BottomChan(Edge As Cell())
    Public Asynchronous LeftChan(Edge As Cell())
    Private Sub Send()
      ... ' Copy computed edge values from grid into buffers
      Up.BottomChan(TopBuff) : Right.LeftChan(RightBuff)
      Down.TopChan(BottomBuff) : Left.RightChan(LeftBuff)
    End Sub
    Private Synchronous Receive()
    Private Sub CaseReceiveAndEdges _
        (TopEdge As Cell(),RightEdge As Cell(), BottomEdge As Cell(), LeftEdge As Cell()) _
        When Receive, TopChan, RightChan, BottomChan, LeftChan
      TopBuff = TopEdge : RightBuff = RightEdge
      BottomBuff = BottomEdge : LeftBuff = LeftEdge
      ... ' Copy received exterior edge value from buffers into grid
   End Sub
End Class
```

Figure 11. Nodes of a generic parallel cellular automata class (outline).

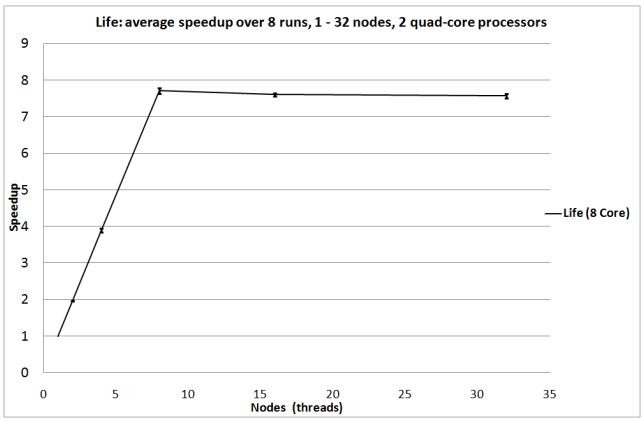

Figure 12. Speedup of Life on a dual, quad-core processor using N nodes executing in parallel.

Running an instrumented version of this code on a work station with 2 quad-core 2.33MHz Intel Xeon chips and 3GB of memory (running 32-bit Vista) shows, unsurprisingly, near linear speedups as we up the number of nodes (keeping the overall grid size fixed), up until we exceed the 8-core count. The graph in Figure 12 plots the ratio between running times of a 1-node sequential and N-node parallel simulation, each computing a 2048 x 2048 cell grid for 1000 iterations. The nodes are arranged horizontally, halving the subgrid width as we double N. Running times are wall-clock times measured from a master thread as the time between starting the simulation and receiving a final "done" signal from the last node to finish. The graph plots the speedups averaged over 8 runs, with standard deviation marked using error bars. For a fair comparison, the single node simulation is modified to avoid the cost of self-synchronization.

Figure 13 shows a screenshot of a (slower) animated version - notice the utilization of all 8 cores and that neighbouring nodes may be computing adjacent generations (shown centred in each node).

4. Compilation

The CB compiler uses the author's `Joins` library (14; 15) as a runtime. Consequently, adding join patterns to Microsoft's production VB 9.0 compiler (implemented in C++) required relatively modest changes, mostly to the front-end of the compiler. Indeed, the implementation closely follows the current implementation of events and declarative event handling. All code is generated as intermediate *source* code for "synthetic" members of various kinds, using pre-existing compiler support. The body of each synthetic member is later compiled to MSIL bytecode by the back end. The synthesized source code relies heavily on type inference and would be tedious to emit directly as explicitly typed MSIL.

4.1 The `Joins` Runtime Library

This section gives a brief overview of the supporting `Joins` library, with enough detail to support the translation process described in later sections. We omit some library features that are not exploited in the translation. This section is adapted from (14), which describes the full API and its implementation in more detail.

In the `Joins` library, the scheduling logic associated with a CB module, class or struct has a separate, first-class representation as an object of a special `Join` class. It is best to think of a join object as a mini-scheduler (or glorified lock) responsible for managing its own set of channels and patterns. Instead of representing channels as special methods belonging to some type, in the library, channels are special delegate values (first-class methods) obtained from some common `Join` object. Communication and/or synchronization takes place by invoking these delegates, passing arguments and optionally waiting for return values. In the library, a join pattern is some code whose execution is guarded by a (linear) combination of distinct channel delegates owned by the same `Join` object. The continuation of a join pattern is provided by the user, not as a method as in CB, but as a delegate. The continuation delegate is free to manipulate resources external to the `Join` object simply by accessing the continuation's private target object.

Users of `Joins` reference the assembly

`Microsoft.Research.Joins.dll`

and import the namespace `Microsoft.Research.Joins`.

A new `Join` instance j is allocated by calling a factory method:

`Join.Create`$($*size*, *allowRedundantPatterns*$)$

The integer *size* bounds the number of channels supported by j; it also sets the constant property j`.Size`. The boolean *allowRedundantPatterns* determines whether the join object accepts or rejects patterns declared on overlapping sets of channels. Its value affects the scheduling of patterns (a round-robin scheduler is used if *allowRedundantPatterns* is `True`; otherwise, a default first-match scheduler is used). CB's compiler always passes `True` to allow overlapping patterns.

A `Join` object notionally owns a set of asynchronous and synchronous *channels*, each obtained by calling an overload of method `Initialize`, passing the location of a *channel* using a VB call-by-reference (`ByRef` or C$^{\#}$ `out`) argument:

j`.Initialize`$($*channel*$)$

Channels are instances of the following delegate types, summarized by a simple grammar of type expressions:

$($`Asynchronous`$|$`Synchronous`$[($`Of` $R$$)])$.`Channel`$[($`Of` $A$$)]$

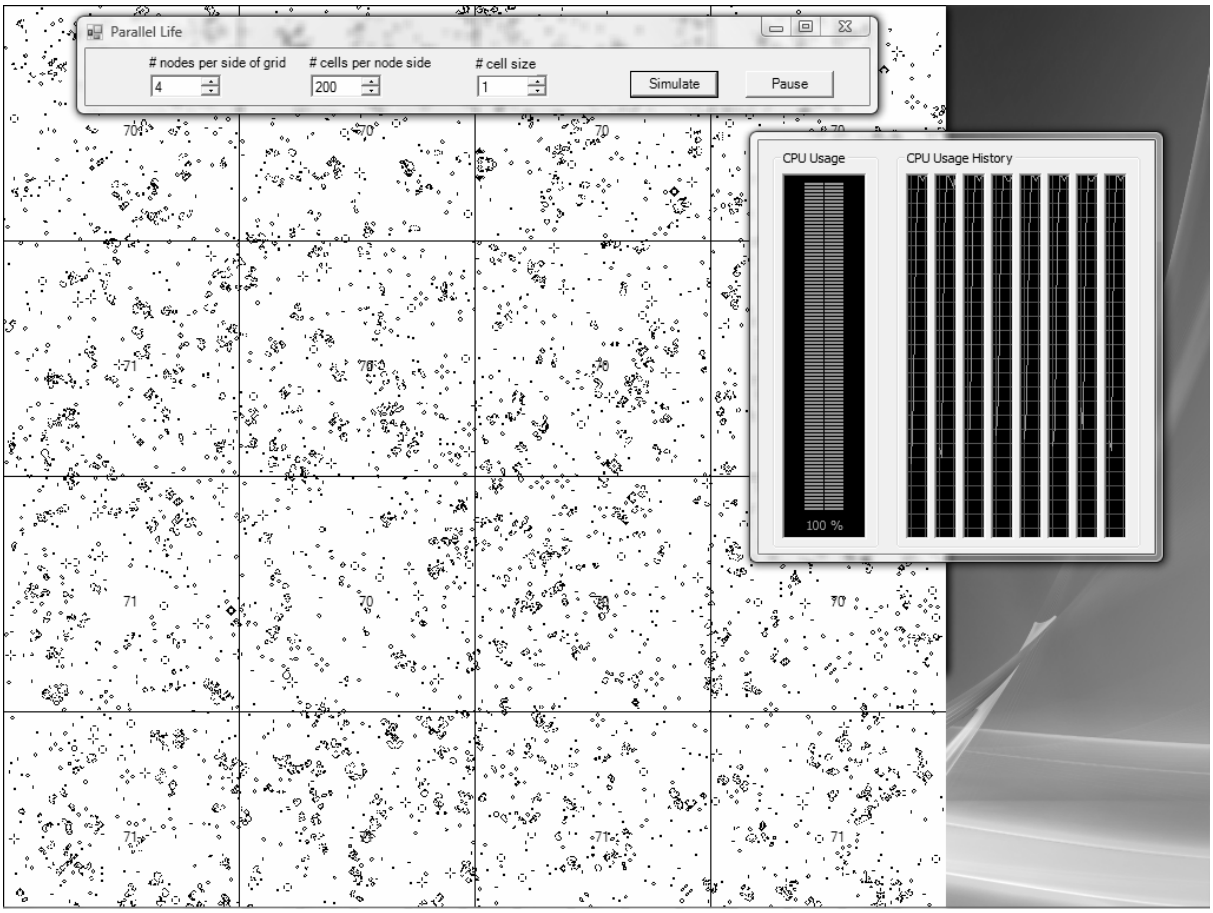

Figure 13. Screenshot of the animated, parallel Game of Life (at reduced resolution for printing).

A channel's outer class, `Asynchronous`, `Synchronous` or `Synchronous(Of` R), should be read as a modifier that specifies its blocking behaviour and optional return type R. Type A, if present, determines the channel's optional argument type. The six channel flavours support zero or one arguments of type A and zero or one results of type R. Multiple arguments or results must be passed in tuples, using the provided generic `Pair(Of` A, B) struct or by other means.

Apart from its channels, a `Join` object notionally owns a set of *join patterns*. A join pattern is constructed by invoking an overload of the instance method `When` followed by zero or more invocations of instance method `And` (or `AndPair`), followed by a final invocation of instance method `Do`. A constructed join pattern typically takes the form:

$$j\texttt{.When}(a_1)\texttt{.And}(a_2)\cdots\texttt{.And}(a_n)\texttt{.Do}(d)$$

where a_1 through a_n are channel delegates and d is a continuation delegate. Argument a_1 of `When`(a_1) may be a synchronous or asynchronous channel or an array of asynchronous channels. Each subsequent argument a_i to `And`(a_i) (for $i > 1$) must be an asynchronous channel; it cannot be a synchronous channel. The argument d to `Do`(d) is a *continuation* delegate that defines the body of the pattern. Although its precise type varies with the pattern, the continuation always has a delegate type of the form:

`Delegate Function` _
`Continuation`$(p_1$ `As` $P_1, \ldots, p_m$ `As` $P_m)$ `As` R

or

`Delegate Sub` _
`Continuation`$(p_1$ `As` $P_1, \ldots, p_m$ `As` $P_m)$

The precise type of the continuation d, including its arity or number of arguments m, is determined by the sequence of channels guarding it. If the first argument a_1 in the pattern is a synchronous channel with return type R, then the continuation is a function with return type R; otherwise the continuation is a subroutine.

The continuation receives the arguments of the joined channel invocations as delegate parameters p_1 `As` P_1, …, p_m `As` P_m, for $m \leq n$. The presence and types of any additional parameters p_1 `As` P_1 …, p_m `As` P_m varies according to the type of each argument a_i joined with invocation `When`(a_i)/`And`(a_i) (for $1 \leq i \leq n$):

- If a_i is of type `Channel` (regardless of flavour), then `When`(a_i)/`And`(a_i) adds no parameter to delegate d.
- If a_i is of type `Channel(Of` P`)` (regardless of flavour), then `When`(a_i)/`And`(a_i) adds one parameter p_j of type $P_j = P$ to delegate d.

Parameters are added to d from left to right, in increasing order of i. A continuation can receive at most $m \leq max$ parameters ($max = 8$ in the current implementation). If necessary, it is possible to join more than max generic channels by calling method `AndPair`(a_i) instead of `And`(a_i). `AndPair`(a_i) modifies the last argument of the new continuation to be a pair consisting of the last argument of the previous continuation and the new argument contributed by a_i.

Readonly property j`.Count` is the current number of channels initialized on j; it is bounded by j`.Size`. Any invocation of j`.Initialize` that would cause j`.Count` to exceed j`.Size` throws `JoinException`. Join patterns must be well-formed, both individually and collectively. Executing `Do`(d) to complete a join pattern will throw `JoinException` if d is `null`, the pattern repeats an asynchronous channel (i.e. is non-linear), an (a)synchronous channel is `null` or *foreign* to this pattern's `Join` instance, or the join pattern is *redundant* (and *allowRedundantPatterns* was `False`). A channel is foreign to a `Join` instance j if it was not allocated by some call to j`.Initialize`. A pattern is redundant when the set of channels joined by the pattern subsets or supersets the channels joined by another pattern on this `Join` instance.

The code generated by CB will, in fact, never raise a `JoinException`: CB's syntactic restrictions and static checks ensure this. A dedicated CB runtime library could avoid checking for these errors and provide a leaner, more tailored API than `Joins` does.

The implementation described here differs only slightly from the one in (14). The bitmask-indexed tables of patterns used to find pattern matches (called `Actions` and `Patterns` in (14, Section 5.1)) are now cyclic instead of null-terminated linked lists. Our support for overlapping patterns, exploited by CB but not described in (14), is implemented simply by advancing the pointer to `Actions` or `Patterns`, respectively, to the node following the selected one, each time an enabled action or pattern is selected for execution (leaving the pointer unchanged if there is no match). Since this round-robin strategy does not, as far as we are aware, guarantee any formal fairness property, users should not assume that competing patterns are executed fairly.

4.2 Basic Translation

To give a taste of the source to source transformation from CB to vanilla VB using the `Joins` library, consider the `Buffer` module from the introduction (Figure 2). The translation performed by the CB compiler yields the code in Figure 14. The `Put` channel is compiled to a pair of a method named `Put` and a field named `PutChannel` that contains the actual `Joins` library asynchronous channel. The compiler-generated body of the `Put` method invokes the delegate `PutChannel` to enqueue its argument. The `<AsynchronousChannel()>` attribute on method `Put` records that it is a channel to support separate compilation, so that, for instance, a subclass can register patterns on an instance channel from an imported base class.

Similarly, the `Take` channel is compiled to a pair of a method named `Take` and a field named `TakeChannel` that contains the actual `Joins` library synchronous channel. The compiler-generated body of `Take` invokes the delegate `TakeChannel` to enqueue its request. Again, the attribute `<SynchronousChannel()>` on `Take` aids separate compilation.

Each join pattern is compiled to a pair of methods: the original continuation method (e.g. `CaseTakeAndPut`), and a new wrapper method (e.g. `CaseTakeAndPutContinuation`) that is the actual continuation registered with the `Joins` library. Although redundant in this simple example, the `xxxContinuation` method for source method `xxx` typically receives all of the arguments from its channels as a single tuple. The n components of this tuple are then forwarded as separate arguments to the source continuation `xxx`.

Finally, the compiler adds a field, `SharedJoin` that holds the `Join` object for the module. The `Join` object itself is initialized by a call to `Join.Create(2,True)`. The size argument, 2, specifies the number of channels required. The `True` argument tells the `Join` object to allow overlapping patterns. A synthetic shared constructor calls the generated method `SharedJoinInitialize()`. This method takes the location of each channel (as a `ByRef` argument) and uses the `Join` object to initialize them. It then registers each generated `xxxContinuation` method (one per source join pattern) with the `Join` object as the continuation delegate of a pattern constructed from the generated channel delegates. Although the channel fields are declared `Readonly` (to prevent direct or indirect malicious updates), since the call to `SharedJoinInitialize()` is from within a constructor, this particular call can take the addresses of our `Readonly` fields without violating the VB or CLR type system.

4.3 Compiling Inheritance

The translation for a class with instance channels and patterns is slightly more involved: a subclass can extend both the set of channels and the set of patterns declared in that class. Our first problem is that when compiling a class, the compiler will only know the size of the `Join` object to allocate for an instance of that class, but not for any of its subclasses. Our second problem is that a subclass may declare additional patterns involving both its own and any inherited channels, all of which must be properly initialized before the patterns are registered on the `Join` object.

Our solution to these problems relies on virtual dispatch and base calls. The class that declares the first channel (and is thus uppermost in the inheritance hierarchy) declares a protected field, called `Join`, of library type `Join`. This field

```
Module Buffer
  Private ReadOnly PutChannel As [Asynchronous].Channel(Of String)
  <AsynchronousChannel()> _
  Public Sub Put( t As String)
    PutChannel(t)
  End Sub
  Private ReadOnly TakeChannel As [Synchronous](Of String).Channel
  <SynchronousChannel()> _
  Public Function Take() As String
    Return TakeChannel()
  End Function
  Private Function CaseTakeAndPut( t As String) As String
    Return t
  End Function
  Private Function CaseTakeAndPutContinuation( Arg As String) As String
    Return CaseTakeAndPut(Arg)
  End Function
  Private ReadOnly SharedJoin As Join = Join.Create(2, True)
  Sub New()
     SharedJoinInitialize(TakeChannel, PutChannel)
  End Sub
  Private Sub SharedJoinInitialize( _
      ByRef TakeChannel As [Synchronous](Of String).Channel, _
      ByRef PutChannel As [Asynchronous].Channel(Of String))
    SharedJoin.Initialize(TakeChannel)
    SharedJoin.Initialize(PutChannel)
    SharedJoin.When(TakeChannel).And(PutChannel).Do(AddressOf CaseTakeAndPutContinuation)
  End Sub
End Module
```

Figure 14. Generated code for the buffer module.

is initialized with `Join.Create(JoinSize(),True)`. The compiler-generated `JoinSize()` method that is `Protected`, but `Overridable`, computes the size (number of channels) of the `Join` object. Each derived class that declares a new instance channel must override the virtual `JoinSize()` method to return the required number of channels for that class, expressed as an increment of `MyBase.JoinSize()`. Since the `Join` field is `Protected`, each class that declares new channels or patterns can reference it to initialize any channels and/or construct any patterns. When necessary, this is handled by a private, compiler generated `JoinInitialize()` instance method, called from a (possibly synthetic) constructor method. The usual chaining of constructor calls ensures that the channels of a base class are properly initialized before a derived class can attempt to add patterns involving them.

For a concrete example, the `ActiveObject` and derived `Person` class compile to the code in Figures 15 and 16. Notice that only `ActiveObject` declares the protected `Join` field but that `Person` overrides the `JoinSize()` method to calculate the size of the `Join` field from `MyBase.JoinSize()`. It also obtains a private `JoinInitialize()` method to perform its own channel initialization and pattern construction. In this way, calling `New ActiveObject()` creates a `Join` object of size 3 (for channels `Start`, `Halt` and `ProcessMessage`), but calling `New Person()` creates a `Join` object of size $3 + 1 + n$ (n is for channels declared in `Person` but not shown).

4.4 Compiling Multi-Parameter Channels and Continuations

The final wrinkle in the translation is dealing with channels and continuations that take multiple parameters. The first problem is that the `Joins` library only supports channels that take zero-or-one parameters. The second is that the library has a ceiling (max) on the number of arguments that may be bound in a continuation. Fortunately, both limitations are easily avoided by tupling arguments before sending them on a channel and by untupling arguments before passing them to a join pattern continuation. Tuples are constructed as nested values of a generic `Pair(Of A,B)` structure, not a class, avoiding some allocation.

5. Extensions

5.1 Customizing Dispatch via Attributes

As described so far, the semantics of CB is that:

1. the continuation of a synchronous pattern runs in the thread of the synchronous sender;

```
Public Class ActiveObject
  Private Done As Boolean
  Protected ReadOnly ProcessMessageChannel As [Synchronous].Channel
  <SynchronousChannel()> _
  Protected Sub ProcessMessage()
    ProcessMessageChannel()
  End Sub
  Protected ReadOnly StartChannel As [Asynchronous].Channel
  <AsynchronousChannel()> _
  Public Sub Start()
    StartChannel()
  End Sub
  Protected ReadOnly HaltChannel As [Asynchronous].Channel
  <AsynchronousChannel()> _
  Public Sub Halt()
    HaltChannel()
  End Sub
  Private Sub CaseStartContinuation()
    CaseStart()
  End Sub
  Private Sub CaseStart()
    While Not Done : ProcessMessage() : End While
  End Sub
  Private Sub CaseHaltContinuation()
    CaseHalt()
  End Sub
  Private Sub CaseHalt()
    Done = True
  End Sub
  Protected Overridable Function JoinSize() As Integer
    Return 3
  End Function
  Protected ReadOnly Join As Join = Join.Create(JoinSize(), True)
  Private Sub JoinInitialize(ByRef ProcessMessageChannel As [Synchronous].Channel, _
                             ByRef StartChannel As [Asynchronous].Channel, _
                             ByRef HaltChannel As [Asynchronous].Channel)
    Join.Initialize(ProcessMessageChannel)
    Join.Initialize(StartChannel)
    Join.Initialize(HaltChannel)
    Join.When(StartChannel).Do(AddressOf CaseStartContinuation)
    Join.When(ProcessMessageChannel).And(HaltChannel).Do(AddressOf CaseHaltContinuation)
  End Sub
  Sub New()
    JoinInitialize(ProcessMessageChannel, StartChannel, HaltChannel)
  End Sub
End Class
```

Figure 15. Generated code for the `ActiveObject` base class with instance channels.

```
Public Class Person
  Inherits ActiveObject
  Private floor As Floor
  Protected ReadOnly GotoFloorChannel As [Asynchronous].Channel(Of Floor)
  <AsynchronousChannel()> _
  Public Sub GotoFloor( f As Floor)
    GotoFloorChannel(f)
  End Sub
  Private Sub CaseProcessMessageAndGotoFloorContinuation( Arg As Floor)
    CaseProcessMessageAndGotoFloor(Arg)
  End Sub
  Private Sub CaseProcessMessageAndGotoFloor( f As Floor)
    floor = f : Dim a As New Ack() : floor.PersonArrived(Me, a) : a.Receive() : ChooseDir(floor)
  End Sub
  Protected Overrides Function JoinSize() As Integer
    Return (MyBase.JoinSize() + 1 + n) ' n is the number of other channels declared in Person
  End Function
  Private Sub JoinInitialize(ByRef GotoFloorChannel As [Asynchronous].Channel(Of Floor), ...)
    MyBase.Join.Initialize(GotoFloorChannel)
    ... ' Initialize other n channels
    MyBase.Join.When(MyBase.ProcessMessageChannel).And(GotoFloorChannel). _
      Do(AddressOf CaseProcessMessageAndGotoFloorContinuation)
    ... ' construct remaining local patterns
  End Sub
  Public Sub New()
    JoinInitialize(GotoFloorChannel)
    ...
  End Sub
  ...
End Class
```

Figure 16. Generated code for the `Person` derived class with a new channel and pattern.

2. the continuation of an asynchronous pattern runs in a newly spawned thread.

One criticism of this design is that it integrates poorly with other threading models available on the platform such as the CLR `ThreadPool`, `Windows.Forms` event loops and Microsoft's new parallel task library (10). Even disregarding other models, spawning a new thread is often unnecessary when the body of an asynchronous pattern is known to execute quickly. Examples of this are when the body just does some trivial calculation before posting a message on another asynchronous channel or when the body just queues a task to some task library. Ideally, the compiler should be able to identify and optimize these cases using some static analysis. However, until such an analysis materialises, it might be preferable to simply let the programmer annotate patterns to indicate how they should be dispatched. The only question is how to surface this in the syntax, without detracting from the pleasing similarity with declarative event handlers.

This section describes a simple proposal that addresses these issues in a semi-declarative yet extensible manner. The idea is that instead of enlarging the syntax of patterns to accommodate some fixed set of options, we instead allow the programmer to use user-defined custom attributes to control the execution *aspect* of individual patterns. Although attributes are typically used to define custom metadata, here we (ab)use them to provide both data and behaviour. This experimental feature turns out to be surprisingly useful.

To support this extension, we extended the `Joins` library to expose a well-known (abstract) custom attribute class, `ContinuationAttribute`, that provides two virtual methods (Figure 17). Method `BeginInvoke(task)` should execute its continuation argument asynchronously (somehow). Method `Invoke(task)` should execute its continuation argument synchronously (somehow).[5] A user may define derived classes of `ContinuationAttribute` and then use them to specify the execution behaviour of individual patterns, simply by annotating a pattern with an instance of the derived attribute. An attribute on a type implicitly applies to each pattern in its definition, giving a simple way to specify default behaviour. The CB compiler recognizes derived instances of `ContinuationAttribute` placed on modules, types or methods and squirrels away a fresh instance of the attribute for each pattern that specifies one. The runtime library then uses

[5] The names `BeginInvoke` and `Invoke`, though odd, follow .NET conventions.

```
Public MustInherit Class ContinuationAttribute
  Inherits Attribute
  Public MustOverride Sub BeginInvoke( task As Continuation)
  Public MustOverride Sub Invoke( task As Continuation)
End Class
Public Delegate Sub Continuation()
```

Figure 17. The abstract `ContinuationAttribute` class.

the attribute to (a)synchronously dispatch the pattern's continuation.

For the optimization scenario, where the user wants to execute a *quick* asynchronous pattern immediately, she might decorate the method with an instance of attribute `Immediate()` (Figure 18).

An example of `ImmediateAttribute`'s use is the generic `Cell(Of T)` class in Figure 19 that supports an asynchronous `Write(v)` and a synchronous `Read()` channel. The operations are rendered atomic by joining each with a private `Token()` message (acting as a lock). Since `CaseWriteAndToken` just stores its value before reposting `Token()`, we can execute it immediately in the last thread to issue `Write` or `Token` instead of spawning a transient thread.

If the user wishes to run asynchronous patterns in the CLR's built-in `ThreadPool`, to save creating new OS threads, she can define the `ThreadPoolAttribute` class in Figure 20. In the following example, the `<ThreadPool()>` attribute ensures that `CaseStartAndIdle` is executed in the CLR `ThreadPool` in response to a `Start` and `Idle` message, without the cost of spawning a new thread each time.

```
Asynchronous Start()
Asynchronous Idle()
<ThreadPool()> _
Private Sub CaseStartAndIdle() When Start, Idle
 ' do some work
 Done(Not cancelled)
End Sub
```

Similarly, to ensure that a continuation is marshalled back to the `Windows.Forms` user-interface thread, she might define the `UI` attribute in Figure 21, and use it as follows:

```
Asynchronous Done( completed As Boolean)
<UI()> _
Sub CaseDone( completed As Boolean) When Done
  Status.Text = _
    If(completed, "Completed", "Cancelled")
End Sub
```

The `<UI()>` attribute ensures `CaseDone(b)` is executed asynchronously when `Done`, but on the UI thread, where it is safe to modify the state of the control `Status`. Note that, without the `<UI()>` attribute, `CaseDone` would be executed in a new thread which would then have to marshal the modifications back to the UI thread using the form's eponymous, but weakly typed, `BeginInvoke(d As Delegate)` method. Here's one of several ugly alternatives using plain CB:

```
Asynchronous Done( completed As Boolean)
Sub CaseDone( completed As Boolean) When Done
  BeginInvoke(Function() CaseDoneBody(completed))
End Sub
Function CaseDoneBody( completed As Boolean)
  Status.Text = _
    If(completed, "Completed", "Cancelled")
End Function
```

This is expensive, type-unsafe and clumsy: `Delegate` is the base class of all delegates, so `d` is invoked by Reflection; VB's lack of lambda-statements forces us to wrap our statement in a function, called from a lambda-expression.

Custom patterns are supported by the CB compiler by copying any attribute that is placed on a pattern `xxx` and derives from `ContinuationAttribute` to its corresponding `xxxContinuation` method. The `Joins` library has been modified to allocate a fresh instance of the attribute, if any, when a pattern is constructed from an `xxxContinuation` method. The library currently uses Reflection to retrieve the attribute instance from a continuation delegate's target method or its declaring class or module. This is expensive and requires some security permissions. However, since attributes are statically known, a better implementation could shortcut Reflection by allocating the required attributes in the `(Shared)JoinInitialize` method and passing them on to a modified `Joins` library.

5.1.1 Example

Figure 22 contains a more realistic example, a form that uses the thread pool to execute a cancellable background task. The task executes concurrently, perhaps in parallel, periodically updating the form asynchronously to indicate progress and final completion. This is a simplified version of what the .NET Framework's existing `BackgroundWorker` class accomplishes.

The code assumes that class `Form` declares four controls: two buttons, `Go` and `Cancel`, a `ProgressBar` and a `Status` label. Clicking `Go` starts one `ThreadPool` thread that asynchronously does some (presumably expensive) work in a loop, updating the `Form`'s `ProgressBar` until `Cancelled` or `Done`. Clicking `Cancel` sets the `Cancellation` cell to `True`. The task polls the cancellation cell to continue working or exit prematurely. Since there is a race between sending a cancellation signal and the task completing, the task's `Done(completed)` message reports the actual comple-

```
Class ImmediateAttribute
  Inherits ContinuationAttribute
  Public Overrides Sub BeginInvoke( task As Continuation)
    task()
  End Sub
  Public Overrides Sub Invoke( task As Continuation)
    task()
  End Sub
End Class
```

Figure 18. Executing quick asynchronous tasks immediately.

```
Class Cell(Of T)
  Private value As T
  Public Asynchronous Write(v As T)
  Public Synchronous Read() As T
  Private Asynchronous Token
  <Immediate()> _
  Private Sub CaseWriteAndToken( value As T) When Write, Token
    Me.value = value : Token()
  End Sub
  Private Function CaseReadAndToken() As T When Read, Token
    Dim value = Me.value : Token() : Return value
  End Function
  Sub New( value As T)
    Me.value = value : Token()
  End Sub
End Class
```

Figure 19. An asynchronous write, synchronous read cell class.

```
Class ThreadPoolAttribute
  Inherits ContinuationAttribute
  Public Overrides Sub BeginInvoke( task As Continuation)
    ThreadPool.QueueUserWorkItem(AddressOf task.Invoke)
  End Sub
  Public Overrides Sub Invoke( task As Continuation)
    task()
  End Sub
End Class
```

Figure 20. Delegating asynchronous tasks to the CLR `ThreadPool`.

```
Class UIAttribute
  Inherits ContinuationAttribute
  Private SC As System.Threading.SynchronizationContext = _
    System.Threading.SynchronizationContext.Current()
  Public Overrides Sub BeginInvoke( task As Continuation)
    SC.Post(Function(state As Object) task(), Nothing)
  End Sub
  Public Overrides Sub Invoke( task As Continuation)
    SC.Send(Function(state As Object) task(), Nothing)
  End Sub
End Class
```

Figure 21. Marshalling asynchronous tasks to the UI thread.

```
Public Class Form
  Inherits System.Windows.Forms.Form
  Private Asynchronous Start()
  Private Asynchronous Idle()
  Private Asynchronous Done(completed As Boolean)
  Private Asynchronous Progress( i As Integer)
  Private Synchronous Await()
  Private Cancelled As New Cell(Of Boolean)(False)
  <ThreadPool()> _
  Private Sub CaseStartAndIdle() When Start, Idle
    Dim cancelled = False
    For i As Integer = 1 To 100
      cancelled = Me.Cancelled.Read()
      If cancelled Then Exit For
      ' Do some work
      Progress(i)
    Next
    Done(Not cancelled)
    Idle()
  End Sub
  <UI()> _
  Private Sub CaseDone( completed As Boolean) When Done
    Cancel.Enabled = False : Go.Enabled = True
    Status.Text = If(completed, "Completed", "Cancelled")
  End Sub
  <UI()> _
  Private Sub CaseProgress( i As Integer) When Progress
    ProgressBar.Value = i
  End Sub
  Sub New()
    InitializeComponent()
    Go.Enabled = True : Cancel.Enabled = False
    Idle()
  End Sub
  Private Sub Go_Click() Handles Go.Click
    Go.Enabled = False : Cancel.Enabled = True : Status.Text = Nothing
    Cancelled.Write(False)
    Start()
  End Sub
  Private Sub Cancel_Click() Handles Cancel.Click
    Go.Enabled = True : Cancel.Enabled = False
    Cancelled.Write(True)
  End Sub
  Public Sub CaseAwaitAndIdle() When Await, Idle
  End Sub
  Protected Overrides Sub OnClosing( e As CancelEventArgs)
    Cancelled.Write(True)
    Await()
    MyBase.OnClosing(e)
  End Sub
End Class
```

Figure 22. Using continuation attributes to control the execution of patterns.

tion reason back to the `Form`. Both the `CaseProgress` and `CaseDone` patterns carry the `UI` attribute since they must be executed on the `Form`'s event loop in order to safely modify its controls. Not that the `Progress` channel is asynchronous to prevent a non-responsive `Form` from blocking the task.

The `Idle()` message merits further explanation. It is used to prevent the form from "closing" while a background task is active. Otherwise, the form could receive a `CaseDone` or `CaseProgress` task *after* its event loop has shut down, causing a run-time error. To avoid this, we override the `OnClosing` method to cancel the task and (synchronously) `Await` the `Idle` token before calling its base method. The `Form` itself is initially `Idle`, but the `Idle` token is consumed and reissued on entry and completion of `CaseStartAndIdle`.

5.2 Synchronous Rendezvous, Revisited

One subtle restriction of CB is that a pattern may mention at most one synchronous channel. This restriction is inherited from its C$^{\#}$ based predecessors. It has the nice property of guaranteeing that the continuation of a synchronous pattern will be run on the thread of its (one and only) synchronous caller. This property is important when the continuation has to run on the same thread as the caller, say to release a lock or access thread local storage. But the guarantee is certainly not always required. Relaxing the restriction would make the syntax of patterns more symmetric and, more importantly, support Ada-style synchronous rendezvous.

Addressing the issue of rendezvous, Cardelli, Benton and Fournet (4)[Section 3.1] propose some syntax (inspired by JoCaml (7)) that allows a pattern's body to "return" to any of several synchronous method headers independently, using a generalized `return` e `to` m`;` statement. The statement returns the value of e to the waiting synchronous caller of m. For example, the following Polyphonic C$^{\#}$ class would allow two threads to exchange values passed to synchronous yet joined methods `f` and `g`.

```
class rendezvous {
  public B f(A a) & public A g(B b) {
     return a to g;
     return b to f;
  }
}
```

Whilst appealing, the authors do not discuss how to handle continuations that fail to return exactly once to each caller, and whether this should be detected statically (as in JoCaml (7)) or dynamically. Instead, we propose a simpler construct that is slightly less expressive, but much easier to explain to users, and no more difficult to implement - in fact, we expect that implementation will be simpler. The idea is to allow multiple channels in a pattern to be synchronous, provided they all have the *same* return type (if any). All synchronous callers involved in a pattern block until the continuation returns one value or exception (the same one) to all of them. This still allows an efficient implementation in which the last caller to enable the pattern can either (if synchronous) immediately execute it without blocking or (if asynchronous) wake up any one waiting caller as appropriate. The drawback is that the continuation can no longer return earlier on some channels than others, forfeiting some concurrency.

As a concrete example, supporting rendezvous would allow us to simplify the code from Section 3.3, removing the need to pass an explicit acknowledgement object, `a`, in the call `floor.PersonArrived(Me,a)`. In the original code, the explicit synchronization after calling `PersonArrived` is needed to avoid a race condition. The person agent needs to be certain that the `floor` has noted its `arrival` before calling `ChooseDir(floor)` to request a lift. So why can't we just make `PersonArrived` synchronous? Unfortunately, object `floor` of class `Floor` (Figure 23) is also an active object that serializes calls to `PersonArrived` using its own synchronous `ProcessMessage` channel. This uses up the one synchronous slot available to the pattern implementing `PersonArrived`, forcing `PersonArrived` to be asynchronous. To work around this restriction, `CaseProcessMessageAndGotoFloor` synchronizes with completion of `PersonArrived(Me,a)` by waiting on `a.Receive()` assuming that `floor` follows protocol and acknowledges with a message `a.Send()`.

However, allowing `PersonArrived` to be synchronous *and* joined with the synchronous `ProcessMessage` lets us simplify the code considerably (Figure 24). The revised `CaseProcessMessageAndPersonArrived` just waits for two synchronous channels with the same (i.e. no) return type. Notice that we have eliminated some complexity (and protocol) from both the caller of and the pattern on `PersonArrived`.

We intend to implement this simple but useful form of rendezvous in future versions of CB.

6. Related Work and Conclusion

Join patterns first appeared in Fournet and Gonthier's foundational join calculus (5; 6), an asynchronous process algebra designed for efficient implementation in a distributed setting. JoCaml (7) and Funnel (13) are functional languages supporting declarative join patterns. Cardelli, Benton and Fournet later proposed an object-oriented version of join patterns for C$^{\#}$ called Polyphonic C$^{\#}$ (3); (4) describes the programming model and an implementation in more detail; while (2) uses Polyphonic C$^{\#}$ to provide a model solution to the *Santa Claus Problem*. Similar extensions to (non-generic) Java, JoinJava, were independently proposed by von Itzstein and Kearney (8). Another implementation of a syntactic variant of Polyphonic C$^{\#}$ was included in the public release of Cω (a.k.a. Comega) in 2004. Cω itself was an extension of C$^{\#}$ 1.1 with both concurrency and, separately, LINQ-like features for SQL and XML data integration (9).

Mostly because of their age, rather than any fundamental restriction, neither Polyphonic C$^{\#}$, Cω nor JoinJava supported Generics, limiting the range of reusable concur-

```
Public Class Floor
  Inherits ActiveObject
  Private people As List(Of Person)
  Public Asynchronous PersonArrived(p As Person,a As Ack)
  Private Sub CaseProcessMessageAndPersonArrived(p As Person,a As Ack) When ProcessMessage, PersonArrived
    people.Add(p)
    a.Send()
  End Sub
  ...
End Class
```

Figure 23. The `Floor` class using emulated rendezvous.

```
Public Class Person
  Inherits ActiveObject
  Private floor As Floor
  Public Asynchronous GotoFloor(f As Floor)
  Private Sub CaseProcessMessageAndGotoFloor(f As Floor) When ProcessMessage, GotoFloor
    floor = f
    floor.PersonArrived(Me)
    ChooseDir(floor)
  End Sub
End Class

Public Class Floor
  Inherits ActiveObject
  Private people As List(Of Person)
  Public Synchronous PersonArrived(p As Person)
  Private Sub CaseProcessMessageAndPersonArrived(p As Person) When ProcessMessage, PersonArrived
    people.Add(p)
  End Sub
  ...
End Class
```

Figure 24. Simplified `Person` and `Floor` classes exploiting rendezvous.

rency abstractions that could be expressed with join patterns. Though essentially for free, CB's combination of join patterns and Generics (also found in JoCaml and Funnel), is hopefully much more compelling.

While JoinJava (8) provides no support for inheritance (concurrent classes must be final), Polyphonic C$^\#$ and Cω had a more subtle inheritance restriction:

> *"(9) If any chord-declaration* [pattern] *includes a virtual method* m [channel] *with the* `override` *modifier, then any method* [channel] n *that appears in a chord with* m *in the superclass containing the overridden definition of* m *must also be overridden in the subclass."(4, Section 3.2).*

Although concisely stated, this condition is arguably difficult to understand. Since the compiler will reject code that fails to override inherited, but conjoined, virtual methods, it effectively forces patterns, not just method signatures, into the interface of a class. But the restriction also has practical ramifications. In our `ActiveObjects` example of Section 3.3 the restriction, if applied, prevents the user from placing the common `CaseHalt()` pattern where it naturally belongs - within the `ActiveObject` base class. Instead, the user is forced to re-declare the pattern within the `Person` subclass and, transitively, within every subclass derived from it. Since the "inherited" pattern accesses private state (i.e. `Done`), that state must now either be revealed as `Protected Done` or be accessed from a revealed method (i.e. `Protected Sub CaseHalt`) in the base class, compromising encapsulation (cf. (4, Section 4.4)).

The CB approach to inheritance, which allows incremental addition of join patterns, seems more natural. CB also makes it easy to override the behaviour of individual patterns, just by marking the declaration of the continuation method as `Overridable` in the base class.

The syntax of Polyphonic C$^\#$ and Cω, allowing a single method to have multiple bodies and a single body to have multiple method headers, is quite alien. The CB design, though verbose, does at least have some resemblance

to an existing language feature: declarative event handling. We hope that this familiarity might ease adoption, by both users and language architects. Nevertheless, one might consider departing from the `Handles`-like syntax to use a parameter binding `When` construct that makes it easier to distinguish channel arguments. Our current syntax, with its explicit method signature, has the advantage of supporting overriding and recursion, which the other syntax does not.

The arrival of Generics in C# 2.0 and VB 8.0 made it possible to encapsulate join pattern constructs in the `Joins` library (15; 14). As well as exploiting Generics in its construction, the library allows users to program generic concurrency abstractions, increasing the utility of join patterns. Compared with CB, a nice feature of `Joins` is that join objects, channels and even partially constructed patterns are first-class values, making it easier to construct higher-level abstractions. The library's main drawback is its reliance on runtime checking to detect what are static errors in CB. While accessible from VB 8.0 and 9.0, VB's lack of support for C#'s anonymous delegates, let alone implicitly typed lambda statements, means that `Joins` remains more awkward to use from VB than C#(even VB 9.0's lambda-expressions are not quite enough).

Our implementation of CB relies on the `Joins` library for runtime support, but it doesn't have to. Although adequate, performance could be further improved by adapting and extending the static compilation techniques originally described for Polyphonic C# and implemented in Cω (see (14) for a comparison between Cω and `Joins`). However, factoring most of the implementation into a runtime library does make it easier to retarget CB to other platforms, by porting the library without modifying the compiler.

One feature we've left out from CB, that was present in Polyphonic C# and Cω, is the introduction of a new `async` *type* as a subtype of `void`. Although useful for specifying asynchronous behaviour in interfaces and on delegate return types, the `async` type must be "erased" to `void` on the underlying platform (the CLR). So, in practice, CB code, expecting to receive an `async` delegate argument, might actually be passed a C# void returning delegate that blocks, violating the expected `async` contract.

Our hope is that CB's natural syntax, support for inheritance, interplay with Generics, and pragmatic extensibility make join patterns viable for inclusion in a future release of Visual Basic; the proposal is now under consideration by Microsoft's Visual Basic team.

Acknowledgments

Thanks to Erik Meijer for suggesting and supporting this work, Harish Kantamneni for detailed code reviews, Danny van Velzen for help with VB sources and Paul Vick, Amanda Silver and the anonymous referees for feedback. Particular thanks to Nick Benton whose examples and prose have been adapted from Polyphonic C# and Cω to `Joins` and CB.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall, 1996.

[2] N. Benton. Jingle bells: Solving the Santa Claus problem in Polyphonic C#, `http://research.microsoft.com/~nick/santa.pdf`, March 2003.

[3] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, number 2374 in LNCS. Springer-Verlag, June 2002.

[4] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26, September 2004.

[5] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 372–385.

[6] C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. In *APPSEM Summer School, Caminha, Portugal, September 2000*, volume 2395 of *LNCS*. Springer-Verlag, 2002.

[7] C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. JoCaml: a language for concurrent distributed and mobile programming. In *Advanced Functional Programming, 4th International School, Oxford, August 2002*, volume 2638 of *LNCS*. Springer-Verlag, 2003.

[8] G. S. Itzstein and D. Kearney. Join Java: An alternative concurrency semantics for Java. Technical Report ACRC-01-001, University of South Australia, 2001.

[9] Microsoft Corporation. Language-Integrated Query (LINQ), `http://msdn2.microsoft.com/en-us/library/bb397926(VS.90).aspx`, 2007.

[10] Microsoft Corporation. Microsoft Parallel Extensions Framework, `http://msdn2.microsoft.com/en-us/concurrency/default.aspx`, 2007.

[11] Microsoft Corporation. Visual Basic Language Specification 9.0 (beta 2), available from `http://www.microsoft.com/downloads`, 2007.

[12] Microsoft Research. Cω, `http://research.microsoft.com/Comega`, 2004.

[13] M. Odersky. An overview of functional nets. In *APPSEM Summer School, Caminha, Portugal, September 2000*, volume 2395 of *LNCS*. Springer-Verlag, 2002.

[14] C. Russo. The Joins Concurrency Library. In Michael Hanus, editor, *Ninth International Symposium on Practical Aspects of Declarative Languages (PADL 2007)*, volume 4354 of *LNCS*, pages 260–274. Springer-Verlag, January 2007.

[15] C. V. Russo. Joins: A Concurrency Library, 2006. Binaries with tutorial and samples: `http://research.microsoft.com/research/downloads`.