CS 15-819: Objects and Aspects: Language Support for Extensible and Evolvable Software

Fall 2004

Typestates for Objects

R. DeLine and M. Fähnrich

In Proceedings of the European Conference on Object-oriented Programming (ECOOP '04), June 2004.

Presented by

Marwan Abi-Antoun

Microsoft .NET Framework

An unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will be shut down immediately.

Value can not be null.
Parameter name: value.

▲ Details | Continue | Quit

```
See the end of this message for details on invoking
just-in-time (JIT) debugging instead of this dialog box.

************** Exception Text **************
System.ArgumentNullException: Value can not be null.
Parameter name: value
   at System.Windows.Forms.ImageCollection.Add(Image value)
   at TypeState.Form1.Form1_Load(Object sender, EventArgs e) in d:\documents\vi
   at System.Windows.Forms.Form.OnLoad(EventArgs e)
   at System.Windows.Forms.Form.OnCreateControl()
```

Main Contributions

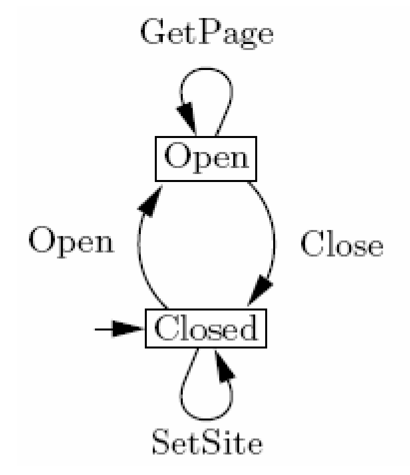
- Extension to notion of conventional type:
  - **Type:** set of operations *ever* permitted on an object
  - **Typestate:** *subset* of operations on an object permitted in a particular context or a particular calling sequence
- A programming model:
  - Detect at compile time using static analysis, nonsensical execution sequences
  - Such defects not detected by type checking or static scoping rules

Main Contributions (continued)

- Novel Object-Oriented support:
  - Supports modern object-oriented languages
  - Composition
  - Sub-classing (downcast, upcast, etc…)
- Performs alias detection and confinement
- Supports extensibility and modularity
- Implementation available Microsoft .NET platform

Problem Statement

- Interfaces have usage rules or protocols
- Typically mentioned in documentation
- Documentation not useful for checking
- Interface rules fall into broad categories:
  - Resource Protocols
  - State Machine Protocols
  - "Calling conventions"
    - Parameter or return value cannot be null

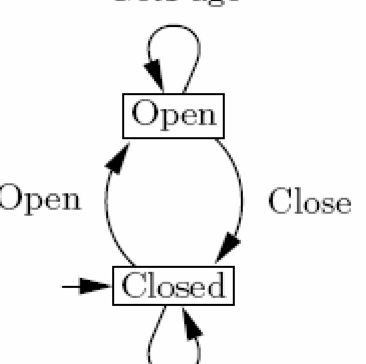


Resource Protocols

- A resource is an object meant to be released through an explicit method call rather than through garbage collection (e.g., a file handle)
- Resource Guarantee:
  - No resource is referenced after its release
  - All resources are released or returned to the method's caller

State-machine protocols

- Limit the order in which an object's methods can be called to the transitions of a finite state machine
  - States have arbitrary symbolic names
  - Transitions between states labeled with method names
- Method Order Guarantee:
  - String of method calls made on an instance that class must be in language accepted by finite state machine



Socket.Send Method (Byte[])

Exceptions

Exception type	Condition
ArgumentNullException	*buffer* is **null**.
SocketException	An error occurred when attempting to access the socket. See the Remarks section for more information.
ObjectDisposedException	The **Socket** has been closed.

Remarks

Send synchronously sends data to the remote host specified in the Connect or Accept method and returns the number of bytes successfully sent. **Send** can be used for both connection-oriented and connectionless protocols.

This overload requires a buffer containing the data you want to send. The SocketFlags value defaults to 0, the buffer offset defaults to 0, and the number of bytes to send defaults to the size of the buffer.

If you are using a connectionless protocol, you must call **Connect** before calling this method, or **Send** will throw a **SocketException**. If you are using a connection-oriented protocol, you must either use **Connect** to establish a remote host connection, or use **Accept** to accept an incoming connection.

If you are using a connectionless protocol and plan to send data to several different hosts, you should use the SendTo method. If you do not use the **SendTo** method, you will have to call **Connect** before each call to **Send**. You can use **SendTo** even after you have established a default remote host with **Connect**. You can also change the default remote host prior to calling **Send** by making another call to **Connect**.

If you are using a connection-oriented protocol, **Send** will block until all of the bytes in the buffer are sent, unless a time-out was set by using System.Net.Sockets.Socket.SendTimeout. If the time-out value was exceeded, the Send. In nonblocking mode, **Send** may complete successfully even if it sends less than the number of bytes in the buffer. It is your application's responsibility to keep track of the number of bytes sent and to retry the operation until the application sends the bytes in the buffer. There is also no guarantee that the data you send will appear on the network immediately. To increase network efficiency, the underlying system may delay transmission until a significant amount of outgoing data is collected. A successful completion of the **Send** method means that the underlying system has had room to buffer your data for a network send.

Note:

If you receive a **SocketException**, use System.Net.Sockets.SocketException.ErrorCode to obtain the specific error code. Once you have obtained this code, you can refer to the Windows Sockets version 2 API error code documentation in MSDN for a detailed description of the error.

Typestates for Objects

- Generalization of original notion[1] of typestate to an abstract *named* predicate over **object graphs**
- An object is in different typestates over its lifetime
- Typestate changes when the object's state (including any nested objects) changes
- Remark: Not to be confused with invariants
- Invariant remains true over the object's lifetime, i.e., an invariant holds in all typestates

1. *Typestate: A Programming Language Concept for Enhancing Software Reliability, by Robert E. Strom and Shaula Yemini*, 1986

Example

WebPageFetcher typestate

```
[TypeStates("Open", "Closed")]
class WebPageFetcher
{
    [Post("Closed"), NotAliased]
    WebPageFetcher([NotNull] string s)...

    [Pre("Closed"), Post("Open"), NotAliased]
    virtual void Open()...

    [Pre("Open"), Post("Closed"), NotAliased]
    virtual void Close()...

    [Pre("Open")]
    [return: NotNull]
    virtual string GetPage([NotNull] string path)...

    [Pre("Closed")]
    virtual void SetSite([NotNull] string site)...

    [NotNull(WhenEnclosingState = "Open")]
    [NotAliased(WhenEnclosingState = "Open")]
    [InState("Connected", WhenEnclosingState = "Open")]
    [Null(WhenEnclosingState = "Closed")]
    private Socket cxn;

    [NotNull]
    private string site;
}
```

Method pre-condition

Method post-condition

Restrict return value

Restrict parameter

Predicates over object graphs: recursive typestate about state of referenced objects

Field object typestate

Invariant (holds in all typestates)

Socket typestate

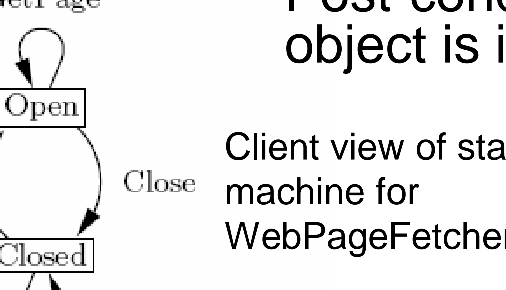
```
[TypeStates("Raw", "Bound", "Connected", "Closed")]
class Socket
{
    [Post("Raw"), NotAliased]
    Socket();

    [Pre("Raw"), Post("Bound"), NotAliased]
    void Bind(string endpoint);

    [Pre("Bound"), Post("Connected"), NotAliased]
    void Connect();

    [Pre("Connected")]
    void Send(string data);

    [Pre("Connected")]
    string Receive();

    [Pre("Connected"), Post("Closed"), NotAliased]
    void Close();
}
```

No aliasing

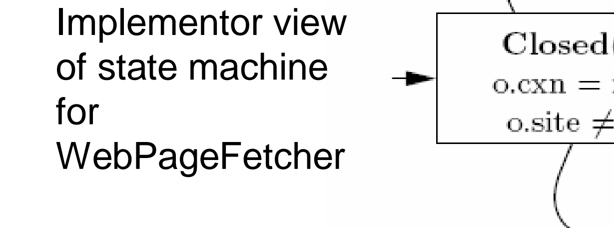
Support for Modularity

- Client view:
  - Predicate uninterpreted function to be matched by name
  - Analysis inspects callee's declaration and not its body
- Implementor view:
  - Predicate defined in terms of predicates over the fields values

- Mechanically verify state machine protocol:
  - Pre-condition: symbolic state the object must be in to call a method;
  - Post-condition is a symbolic state the object is in after the method returns



Client view of state machine for WebPageFetcher



Implementor view of state machine for WebPageFetcher

Notation for Typestates

- Typestate specification applies to class frame first introducing the typestate **name** (via **TypeStates(…)** annotation)
- Without the annotation, objects are assumed to be in state **State.Default**
  - System.Object @ State.Default
- Can be targeted at a particular class frame **C**
  - using **TypeStates(…, Type = C)**
- Can be targeted at all subclasses
  - using **TypeStates(…, Type = Subclasses)**

Special Issues

- Composition
  - Addressed with **[WhenEnclosingState]**
- Subclassing
  - Downcasting, upcasting, virtual dispatch, direct calls
  - Addressed with sliding method signatures
- Null References
  - Addressed with **[Null], [NotNull]**
- Alias detection
  - Addressed with **[NotAliased], [MaybeAliased]**

Composition

- Used when both the object and its fields (referencing nested objects) have state-machine protocols
- State mapping
  - Relates state of an object to the state of its fields:
  - Using [**WhenEnclosingState]** construct
- Ensures the “outer” object is well-behaved client of any nested “inner” objects with their own state machine protocols

Object Typestate Revisited

- **Class frame:** set of fields of the object declared in that particular class, not in any super- or sub-classes.
- **Frame typestate:** constraints on fields in a class frame
- **Object typestate = collection of frame typestates**
  - one per class frame of the object's allocated dynamic type.
  - **rest** typestate (uniform typestate for all unknown subclass frames)

Object
WebPageFetcher
CachingFetcher
⋮

An object decomposed into class frames. The ellipsis reflects that the object's dynamic type may be more derived than its declared type.

Object@s_1
WebPageFetcher@s_2
CachingFetcher@s_3
rest@s_4

A frame typestate per frame ($s_1..s_3$) and one frame typestate (s_4) for statically unknown subclasses.

Downcasting, Upcasting, etc…

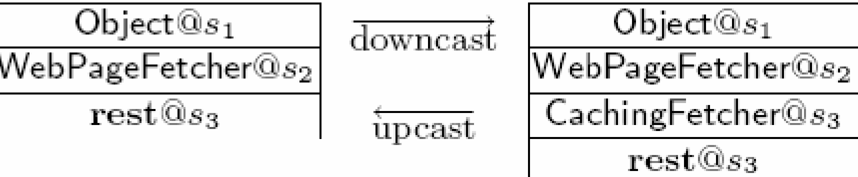
- Downcasting: casting a pointer or reference to a base class, to a derived class.

```
– WebPageFetcher w = new CachingFetcher ();
CachingFetcher f = (CachingFetcher )w;
```

- Upcast: casting a pointer or reference to a derived class to a more general base class

```
CachingFetcher f = new CachingFetcher();
WebPageFetcher w = (WebPageFetcher)f;
```

- Virtual call: a call subject to dynamic dispatch



Typestates and Subclassing -1

```
[TypeStates("CacheOnly")]
class CachingFetcher : WebPageFetcher
{
    [Post("Closed"), NotAliased]
    CachingFetcher(string site) : base( site ) {}
    [Pre("Closed"), Post("Open"), NotAliased]
    override void Open()
    {
        base.Open();
        this.cache = new Hashtable();
    }

    [Pre("Open"), Post("Closed"), NotAliased]
    override void Close()...

    [Pre("Open")]
    [return: NotNull]
    override string GetPage([NotNull] string path)...

    [Pre("Open"), Post("CacheOnly"), Post("Closed", Type = WebPageFetcher), NotAliased]
    void CloseKeepCache()...

    [Pre("CacheOnly"), Pre("Closed", Type = WebPageFetcher)]
    [return: MayBeNull]
    string GetCachedPage(string path)...

    [Pre("CacheOnly"), Pre("Closed", Type = WebPageFetcher), Post("Closed"), NotAliased]
    void DeleteCache()...

    [Null(WhenEnclosingState = "Closed"), NotNull(WhenEnclosingState = "Open,CacheOnly")]
    private Hashtable cache;
}
```

A subclass can associate its own field invariants with its superclasses' typestates

Typestates and Subclassing - 2

```
[TypeStates("CacheOnly")]
class CachingFetcher : WebPageFetcher
{
    [Post("Closed"), NotAliased]
    CachingFetcher(string site) : base( site ) {}
    [Pre("Closed"), Post("Open"), NotAliased]
    override void Open()...

    [Pre("Open"), Post("Closed"), NotAliased]
    override void Close()...

    [Pre("Open")]
    [return: NotNull]
    override string GetPage([NotNull] string path)
    {
        string page = this.cache.GetValue(path);
        if (page == null)
        {
            page = base.GetPage(path);
            // NO NEED to check that field cache is NOT null
            // Guaranteed by WebPageFetcher.GetPage(...)
            // specifying frame typestate "Open" for all subclasses
            this.cache.Add(path, page);
        }
        return page;
    }

     :

    [Null(WhenEnclosingState = "Closed"), NotNull(WhenEnclosingState = "Open,CacheOnly")]
    private Hashtable cache;
}
```

Overriding methods can take advantage of object typestate precondition on the receiver

Typestates and Subclassing - 3

```
[TypeStates("CacheOnly")]
class CachingFetcher : WebPageFetcher
{
    [Post("Closed"), NotAliased]
    CachingFetcher(string site) : base( site ) {}
    [Pre("Closed"), Post("Open"), NotAliased]
    override void Open()
    {
        base.Open();
        this.cache = new Hashtable();
    }

    [Pre("Open"), Post("Closed"), NotAliased]
    override void Close()...

    [Pre("Open")]
    [return: NotNull]
    override string GetPage([NotNull] string path)...

    [Pre("Open"), Post("CacheOnly"), Post("Closed", Type = WebPageFetcher), NotAliased]
    void CloseKeepCache()...

    [Pre("CacheOnly"), Pre("Closed", Type = WebPageFetcher)]
    [return: MayBeNull]
    string GetCachedPage(string path)...

    [Pre("CacheOnly"), Pre("Closed", Type = WebPageFetcher), Post("Closed"), NotAliased]
    void DeleteCache()...

    [Null(WhenEnclosingState = "Closed"), NotNull(WhenEnclosingState = "Open,CacheOnly")]
    private Hashtable cache;
}
```

Subclass can define its own typestates

Typestate targeted at a particular class frame

There is a problem

```
[TypeStates("Open", "Closed")]
class WebPageFetcher
{
    [Post("Closed"), NotAliased]
    WebPageFetcher([NotNull] string s)...

    [Pre("Closed"), Post("Open"), NotAliased]
    virtual void Open()...

    [Pre("Open"), Post("Closed"), NotAliased]
    virtual void Close()...

    [Pre("Open")]
    [return: NotNull]
    virtual string GetPage([NotNull] string path)...

    [Pre("Closed")]
    virtual void SetSite([NotNull] string site)...

    [NotNull(WhenEnclosingState = "Open")]
    [NotAliased(WhenEnclosingState = "Open")]
    [InState("Connected", WhenEnclosingState = "Open")]
    [Null(WhenEnclosingState = "Closed")]
    private Socket cxn;

    [NotNull]
    private string site;
}
```

- **Post** specification states that all frames, including subclasses, will be in typestate **Open** at the end of the method body. But this is not satisfied by the implementation…
- Fields in subclasses are not visible through static scope
- Subclass typestates must still be **Closed**

```
[TypeStates("CacheOnly")]
class CachingFetcher : WebPageFetcher
{
    [Post("Closed"), NotAliased]
    CachingFetcher(string site) : base( site ) {}
    [Pre("Closed"), Post("Open"), NotAliased]
    override void Open()
    {
        base.Open();
        this.cache = new Hashtable();
    }
    ...
```

Corrected WebPageFetcher

```
[TypeStates("Open", "Closed")]
class WebPageFetcher
{
    [Post("Closed"), NotAliased]
    WebPageFetcher([NotNull] string s)
    {
        this.site = s;
    }

    [Pre("Closed"), Post("Open"), Post("Closed", Type = Subclasses)]
    virtual void Open()...

    [Pre("Open"), Post("Closed"), NotAliased]
    virtual void Close()...

    [Pre("Open")]
    [return: NotNull]
    virtual string GetPage([NotNull] string path)...

    [Pre("Closed")]
    virtual void SetSite([NotNull] string site)...

    [NotNull(WhenEnclosingState = "Open")]
    [NotAliased(WhenEnclosingState = "Open")]
    [InState("Connected", WhenEnclosingState = "Open")]
    [Null(WhenEnclosingState = "Closed")]
    private Socket cxn;

    [NotNull]
    private string site;
}
```

Sliding Method

Sliding methods

- A virtual method for which:
  - Overriding methods get to assume same pre condition as base method and have to establish same post condition state as base method
- To change the typestate of subclasses (i.e., entire object typestate)
  - Make a virtual call to a sliding method
  - Dispatches to dynamic type (lowest class frame)
  - Changes the typestate of its frame and all frames of **superclasses**
  - Leaves the **subclass** typestates unchanged

Sliding Method: Case 1

virtual **C**::implSig$_{C}$ ([Pre(*s*),Post(*t*),Post(*r'*,Type = Subclasses)]

override **D**::implSig$_{D}$([Pre(*s*),Post(*t*),Post(*r'*,Type = Subclasses)]

[Sliding]
virtual **C**::virtSig$_{C}$([Pre(*s*), Post(*t*, Type = Subclasses)]

C = WebPageFetcher
D = CachingFetcher
M() = Open()

s = Closed
t = Open
r = *r'* = Closed

Requires **D**.M() to call **C**.M() **before** changing its typestate from *s* to *t*

implSig$_C$: $\begin{array}{|c|}\hline \chi_B^{pre} \\ \hline C@s \\ \hline \mathbf{rest}@r \end{array} \longrightarrow \begin{array}{|c|}\hline \chi_B^{post} \\ \hline C@t \\ \hline \mathbf{rest}@r' \end{array}$

implSig$_D$, $D \ll C$: $\begin{array}{|c|}\hline \chi_B^{pre} \\ \hline C@s \\ \hline \vdots \\ \hline D@s \\ \hline \mathbf{rest}@r \end{array} \longrightarrow \begin{array}{|c|}\hline \chi_B^{post} \\ \hline C@t \\ \hline \vdots \\ \hline D@t \\ \hline \mathbf{rest}@r' \end{array}$

virtSig$_C$: $\begin{array}{|c|}\hline \chi_B^{pre} \\ \hline C@s \\ \hline \mathbf{rest}@s \end{array} \longrightarrow \begin{array}{|c|}\hline \chi_B^{post} \\ \hline C@t \\ \hline \mathbf{rest}@t \end{array}$

Sliding Method: Case 2

virtual **C**::implSig$_C$ ([Pre(*s*),Post(*t*),Post(*r’*,Type = Subclasses)]

override **D**::implSig$_D$([Pre(*s*),Post(*t*),Post(*r’*,Type = Subclasses)]

[Sliding]
virtual **C**::virtSig$_C$([Pre(*s*), Post(*t*)]

C = WebPageFetcher
D = CachingFetcher
M() = Open()

s = Closed
t = Open
r = *r’* = Open

Requires **D**.M() to change its typestate from *s* to *t* **before** call to **C**.M()

implSig$_C$: χ_B^{pre} [$C@s$ | $\mathbf{rest}@r$] $\longrightarrow$ χ_B^{post} [$C@t$ | $\mathbf{rest}@r'$]

implSig$_D$, $D \ll C$: χ_B^{pre} [$C@s$ | $\vdots$ | $D@s$ | $\mathbf{rest}@r$] $\longrightarrow$ χ_B^{post} [$C@t$ | $\vdots$ | $D@t$ | $\mathbf{rest}@r'$]

virtSig$_C$: χ_B^{pre} [$C@s$ | $\mathbf{rest}@s$] $\longrightarrow$ χ_B^{post} [$C@t$ | $\mathbf{rest}@t$]

Non-virtual call site

WebPageFetcher::Open

Before		After
Object@Default		Object@Default
WebPageFetcher@Closed	⟶	WebPageFetcher@Open
rest@Closed		rest@Closed

CachingFetcher::Open

Before		After
Object@Default		Object@Default
WebPageFetcher@Closed		WebPageFetcher@Open
CachingFetcher@Closed	⟶	CachingFetcher@Open
rest@Closed		rest@Closed

Non-virtual call site

```
[TypeStates("Open", "Closed")]
class WebPageFetcher
{
    [Pre("Closed"), Post("Open"), Post("Closed", Type = Subclasses)]

    virtual void Open()...
        ...
```

2. Upcast to WebPageFetcher

Object@Default :: **WebPageFetcher@Closed** :: rest@Closed

3. Direct base-call to WebPageFetcher.Open()

Object@Default :: **WebPageFetcher@Open** :: rest@Closed

```
[TypeStates("CacheOnly")]
class CachingFetcher : WebPageFetcher
{
```

1. Object@Default :: **WebPageFetcher@Closed** :: **CachingFetcher@Closed** :: rest@Closed

```
    [Pre("Closed"), Post("Open"), Post("Closed", Type = Subclasses)]

    override void Open()
    {
        base.Open();
```

4. Downcast back to CachingFetcher

Object@Default :: **WebPageFetcher@Open** :: CachingFetcher@Closed :: rest@Closed

```
        this.cache = new Hashtable();
    }
}
```

5. Post-condition to WebPageFetcher.Open()

Object@Default :: **WebPageFetcher@Open :: CachingFetcher@Open** :: rest@Closed

Virtual call site

- The virtual call to Open changes all frames to typestate **Open**
- Dynamically, every frame of the object is changed

Virtual call site

1. Object@Default :: **WebPageFetcher@Closed** :: **CachingFetcher@Closed** :: rest@Closed

```
[TypeStates("Open", "Closed")]
class WebPageFetcher
{
   [Pre("Closed"), Post("Open"), Post("Closed", Type = Subclasses)]

   virtual void Open()...
      ⋮
```

2. Upcast to WebPageFetcher
Object@Default :: **WebPageFetcher@Closed** :: rest@Closed

3. Virtual call to WebPageFetcher.Open()
Object@Default :: **WebPageFetcher@Open** :: **rest@Open**

4. Downcast back to CachingFetcher
Object@Default :: **WebPageFetcher@Open** :: **CachingFetcher@Open :: rest@Open**

```
WebPageFetcher w = new CachingFetcher ();
w.Open();
```

Extensibility through Subclassing

- Uniform typestate to all unknown subclass frames
- Subclasses can provide new interpretations of typestate for their own fields
- Subclasses only constrain fields in own class frame, and not in any frames of **superclasses**
- Field state change in class frame affects that frame's typestate and does not affect typestates of any **subclass** frames
- Describe intermediate states of objects
  - Part of the object is in state A.
  - Other parts in state B

Aliasing

- A sound static check of object invariants must be aware of all references to an object which can change the object's state
- E.g., following call sequence legal only if `objA` and `objB` are not the same object

```
objA.Open();
objB.Open();
```

- Adopted solution: Method's **declaration** to advertise all objects whose states the method changes

[NotAliased] Parameter

- Guarantees to method (callee) :
  - It can access parameter object through the given parameter or copies of the pointer that the method makes
  - Cannot access object through any other access paths
- Guarantees to caller:
  - Upon return, method will not have any more aliases to object

[NotAliased] Field

- Object with that field holds only pointer to referenced object

Leaking

- An object can leak, i.e., transition to the **MaybeAliased** mode.
- Alias confinement when an object leaks:
  - All references to **[MaybeAliased]** object become typestate invariant
  - Typestate frozen to current typestate
  - Recursively leak, i.e., treat all fields of aliased object as aliased

Formal Model

- Extends typing rules with typestates
- Used to prove the correctness of method implementations such as:
  - overriding method not calling base class method
  - out of order calls
- No formal soundness proof (future work)

Available Implementation

- Microsoft Fugue available for download
  - Supports Microsoft .NET platform (C#)
  - http://research.microsoft.com/research/downloads/default.aspx
- Remark: syntax quite different from what was presented in this paper

Limitations with Approach

- Cannot handle constraints in the form of general relations between fields of different frames or different objects
- Can only express finite-state abstract constraints
  - E.g., only enforce that Pop called on a stack object only after a call to Push or a non-emptiness test
  - Cannot ensure Push is called at least as many times as Pop

Limitations with Approach

- Cannot handle multiple outcomes for operations
  - Normal outcome v/s exceptional outcome (due to exceptions), e.g., unable to obtain resource to complete operation
- Cannot correlate object's typestate to return value from a method
  - See Enumerator example below
- Cannot handle [in/out] arguments
- How to prevent redefinition of typestates in a class hierarchy?

Additional Content: Custom States

- Overcome limitations of state machine protocols (pre- and post-condition).
- Model state of object with another object, called "custom state"
  - Plug-in pre-condition and post-conditions
  - In fact, methods of the custom state object
  - Invoked during checking to perform interface-specific state checks and state transitions
- Example:
  - extend general typestate analysis with domain-specific algorithms, like SQL parsing

IEnumerator Example

- Need to correlate object's typestate to return value from MoveNext()
- Question: Can we really specify typestates on an abstract interface instead of an implementation class? How can we say that when an instance of a class which implements IEnumerator object is created, it must be in state "InBounds"?

```
IEnumerator ie = list.GetEnumerator();

ie.Reset();
while( ie.MoveNext())
{
    Console.WriteLine( ie.Current );
}
```

```
[ TypeStates("InBounds ", " OutOfBounds") ]
interface IEnumerator
{
  [ Pre("InBounds ") ]
  object Current { get; }

  [ Post ("InBounds ", WhenReturnValue=true),
    Post ("OutOfBounds", WhenReturnValue=false) ]
  bool MoveNext ();

  [ Post ("OutOfBounds") ]
  void Reset ();
}
```

Discussion

- Decidability problems with rich logic with preconditions and postconditions and object invariants runs into decidability problems:
  - Is this approach immune?
  - Are custom typestates decidable?
- According to authors, typestates maintain the "simplicity and feel of types". Is this arguable?
- What are parametrized typestates (briefly mentioned in the paper)?

Editorial Opinion

- Encoding interface usage rules in the code is particularly important and useful for reusable software (e.g., components, class libraries, frameworks, etc…)
- The designer cannot how the public methods you expose will be used and you need all the help you can get to protect your code from undefined calling sequences
- The alternative is to spend many hours writing documentation which will soon be out of date.
- Documentation will not be enough and extensive white box testing will still be needed