TECHNICAL WHITE PAPER

# A Thread-Safe Extension to Object-Oriented Programming

*by Gael Fraiteur, Founder & Director at PostSharp Technologies*
gael@postsharp.net

## ABSTRACT

Two generations of research and development in programming languages have provided the robust memory and execution models that everybody now takes for granted. Good programming languages are built on good programming models, and good programming models are close to the way people think. They are simple so they require less cognitive load; they can be verified by the machine so human mistakes are detected sooner; and they allow developers to express intent in a concise way. In short, good programming models require less talent from developers and make them more productive.

Although the memory and execution models have arguably come to maturity, threading models still seem to be in infancy. Some threading models (Immutable, Actor) exist and have received recognition in some niches, but they have failed to get wide adoption in the industry, largely dominated by object-oriented programming.

This paper will formalize the notion and importance of threading models for the future of object-oriented programming and mainstream software development. It proposes a unified object-oriented abstraction and design of threading models and shows how they can be practically applied to Immutable, Freezable, Synchronized, Reader-Writer Synchronized, Actor and Thread Affine models.

It first discusses the requirements that our design puts on the compiler. Then it shows how we implemented threading models with PostSharp, a 100%-compatible compiler extensibility technology that allows building extensions to the C# and VB languages.

However, this is not a tutorial only of our implementation (see PostSharp Threading Pattern Library for reference documentation and tutorials). Our design is described in an implementation-agnostic way and is applicable to other languages and runtime environments.

This approach to thread-safe object-oriented programming is pragmatic and cross-disciplinary. It stems from industry practice, not from academic research. It does not aim at *complete* soundness and verifiability. It combines several technologies including aspect-oriented programming, static analysis and dynamic analysis, to achieve *high* verifiability in the presence of a single-threaded test coverage. It takes the point that "the best is the enemy of the good," and the proposed solution can greatly improve the productivity and reliability of mainstream multi-threaded applications.

This paper is presented as an open contribution to the computer industry with the hope of receiving feedback and provoking improvements in object-oriented languages.

## TABLE OF CONTENTS

# A BIASED HISTORY OF PROGRAMMING LANGUAGES

## Execution and Memory Models

Programming languages, like all human languages, allow the expression of meaning against a model. Words of the programming language map to concepts of the programming model.

For instance, consider the UNIVAC I Assembly Language in 1951, arguably the first commercially used programming language. Like today's assembly languages, it defines instructions that enable moving words from a memory address to a named register, performing arithmetic operations on registers and transferring the control flow to another memory address. It also has instructions to read words from the console or from a tape.

Today, it seems that this instruction set is at a very low level. However, it would be a mistake to conclude that assembly language directly controls the hardware. The instruction set is actually much simpler, by orders of magnitude, than the hardware itself. The UNIVAC memory was implemented using mercury delay lines, where information was actually stored as acoustic impulses. It was not random-access memory by any way, but the hardware implemented an abstraction that made it addressable, so programmers could reason in terms of words and addresses. Likewise, when you wanted to read a block from a tape, you did not have to control the electric motors, and you didn't even have to worry if the tape was paper or magnetic. You programmed against a model of the hardware.

However, the UNIVAC I Assembly Language did operate at a very low level of abstraction. Programmers were already programming against a model, but this model was still quite distant from the ones people used to reason and communicate. They don't normally use memory addresses to identify things.

People use words. Words are symbols for things, originally sequences of sounds that we eventually (quite recently about 5,200 years ago) learned to write. When we think and communicate, we use words, not just numbers. FORTRAN I (1955) attempted to bridge this gap by introducing a brand new concept: global named variables instead of memory addresses. There were two kinds of variables: scalars and arrays. Names could be chosen from a list of predefined ones.

Global variables and arrays were the single decisive innovation of FORTRAN I over UNIVAC assembly language. What FORTRAN I did was to provide a better memory model to program against. There was nothing you could do in FORTRAN that you could not do in assembly language (the opposite was true) but the proposed memory model was closer to the way people think. It took a dozen man-years to implement a compiler for the new language and it multiplied programmers' productivity by an order of magnitude.

If FORTRAN I brought a memory model, FORTRAN II (1958) introduced an execution model. Functions and subroutines are named subprograms, so programmers no longer had to rely on memory addresses to transfer execution. Additionally, the new concepts enabled programmers to better divide a big problem into several small problems and address each small problem individually. Again, there was nothing they could do with FORTRAN II that they could not do in assembly language, but the new execution model was closer to the way the human mind works and productivity increased again.

ALGOL 60 refined the procedural model by introducing the notions of nested blocks and lexical scoping, further enhancing the ability of the language to decompose large problems into smaller independent problems and cope with the limitations of people to reason holistically about a large number of items.

An evolutionary view of programming languages suggests that characteristics that make the language more human-friendly, i.e. require less talent and cognitive load to work with, become successful and are copied to newer languages. The COBOL data structures (1959) made it into all modern languages, but its chatty syntax did not. Lisp (1958) introduced several innovations, but it was the stack that first gained large adoption and it took another 30 years for garbage collection to become mainstream. Its recognizable list-based syntax, cumbersome but mathematically elegant, has never made it into commercial languages.

It is tempting to believe that higher abstraction makes a programming language better, but the fate of Lisp S-expressions proves this theory wrong. Programmers (those people who talk to the machines) value the *right* kind of abstraction, neither too low nor too high. They like programming languages that allow them to express, in source code, models of the reality they need to cope with. An overwhelming majority of people still rely on Newton's physics for their everyday life and job even if Einstein's physics have been proved more accurate. This is because Newton's physics are a more workable model of the world for the limited scope of our daily lives. Only a small niche of scientists and engineers have to take relativist and quantic effects into account. The same holds true for programming models and languages. This phenomena can account for the relative lack of success of pure functional programming languages in most business and web applications.

Simula (1962) is considered the first object-oriented programming language. Simula was designed to execute computer simulations of the physical world. To simulate a process, engineers first build a model and then compute with the model using different inputs. It is remarkable that object-oriented programming was invented by scientists who wanted to create computable models of the world. Object-oriented programming enhances both the memory and execution models by introducing inheritance and method overriding. From an evolutionary perspective, Simula was very successful and is the direct ancestor of C++, Java and C#.

## Qualities of Good Programming Models

If you were hibernating between 1955 and 1995, you would surely be amazed by the progress made by programming languages. You would probably value the following qualities:

- **Adequate Abstraction**. There is a unified concept of "location" and "assignment" so when you think "a=1", you don't have to think whether "a" is a static field, instance field, parameter or local variable. Computing the memory address is completely abstracted from you. If it has store and load semantics, this is a location and this is all you need to know. Also, the concepts of functions and classes allow you to build your own abstractions.
- **Verifiability.** Your code is guaranteed to be type safe. You cannot load from an invalid memory address. You cannot transfer execution to something that is not code. You will never corrupt the memory.
- **Locality and separation of concerns.** Every piece of code is carefully isolated from the others. Defects are much less likely than before to have global effect. You cannot write to a memory address that is actually a parameter of another subroutine. Implementation details of other components are purposefully made inaccessible to you through the "protected" and "private" keywords.
- **Determinism (predictability).** Well-written programs are characterized by a high level of determinism or predictability. This is principally due to verifiability and locality. Before the invention of local scopes, it was easy to erroneously modify a memory location that affect another procedure because it was not planned that the procedure could be invoked at this specific time.

- **Performance.** The cost of higher abstraction is generally worse performance, and advances in abstraction are often criticized for their performance cost. Performance is not a characteristic of the model itself, but a characteristic of the use of the model in a specific situation.

As a result of these qualities, modern programming languages, compared to their ancestors:

- Allow for higher productivity: developers produce more value per hour;
- Require less skills: developers with less experience, talent and training can become productive;
- Produce fewer errors.

Eventually, the objective of programming language design is to *make it easier to build good software*.

## Threading Models Today

Programming languages were initially built around *memory* models and *execution* models. Languages that share exactly the same models are almost interchangeable, as are Visual Basic .NET and C#. However, these models have been designed for single-processor computers. They do not address the issues raised by multi-processors, such as concurrent execution and synchronization of shared resources.

Concurrency has traditionally been the dominion of operating systems. Notions of processes, threads, events, mutexes and semaphors directly stem from the OS kernel. If we add different kinds of locks, typically implemented by the application framework, we have the set of primitives that application can use to address multithreading: a very low-level one.

For most developers, multithreading is where memory management was in the 1960s: so low-level that it is too cognitively challenging, requires rare skills, consumes too much time, and causes too many defects. Clearly, the industry needs the same evolution as with memory; we need a threading model that is closer to the way human think, not how things are implemented by the hardware or the operating system.

Threading models are similar to memory and execution models in the fact that they offer adequate abstraction, they are machine-verifiable, and they offer allow for good locality. Additionally, threading models should be evaluated against their degree of determinism. Not all threading models score the same on all qualities.

Some threading models have been implemented in niche programming languages. Recently, some of these languages have gained higher popularity, driven by the urge to address the multithreading issue:

- The **Immutable model** is the essence of pure functional programming; it eliminates the complexity introduces by memory assignments, and therefore allows for a very high level of verifiability, determinism, and compiler optimizations. However, pure functional programming tends to be an inadequate level of abstraction for most business, web and mobile applications, and requires more skilled and expensive developers.
- The **Actor model** divides computing across entities named “actors”. Each actor has its own memory space and cannot access the one of its neighbors. Actors communicate with each other using immutable messages. Each actor has its own message queue and a single thread to process the queue. Because the only mutable state is private to an actor and all shared state is immutable, there is no concurrency issue. Although the Actor model can be implemented as a class library for virtually any programming language, verifiability and abstraction can be achieved only with

proper support from the programming language. Erlang is the most famous implementation of the Actor model.

- The **Transactional model**, albeit despised by many application developers, has probably the widest commercial adoption of all concurrency models. It is implemented in transactional databases where most business data is still being stored. Software transactional memory is a generalization of database transactions to general application development. Research implementations as class libraries are available for most programming languages, but like the Actor model, the full benefits of using a threading model can only be leveraged if the feature is integrated into the language.

Although threading models have been in use for several decades, they still have not fully made it into mainstream application development. Part of the reason is that these threading models require developers to choose between switching to a different programming language and using a class library, often with research-level maturity, which does not deliver the same guarantees as a language-integrated feature.

## TARGET SCOPE AND AUDIENCE OF THREADING MODELS

Abstraction has often been considered harmful to multi-core programming. A first common argument is that abstraction hinders performance. According to a second argument, abstraction obfuscates the real semantics of the program, making them harder to reason about. In case that it is necessary to reason at a low level of abstraction, the coexistence of constructs of high and low levels of abstraction makes the task more difficult.

Both arguments are valid in their particular context. When implementing core data structures, performance is absolutely crucial. For instance, operating system kernels and concurrent collections in base class libraries must be highly optimized because they are the layer on which all other pieces of software will run. It makes sense, in this case, to set up a team of several computer science PhDs during a few months or years. When high performance is required, it is necessary to think at a low level of abstraction. Not only higher abstraction hinders performance, but it also harms understanding.

Note that the same arguments were used at each generation of programming languages. When FORTRAN I was introduced, some programmers believed they could achieve better memory utilization in hand-written assembly, and refused to use the new language. Some people still prefer C over C++ for similar reasons. When Java and C# were proposed, many feared the lack of control. C and C++ are still preferred for performance-critical applications. So, at each generation, people debate about control versus abstraction, or performance versus productivity.

Threading models target a very different scope than system programming. Their typical audience are large application development teams.

When a development team is composed of more than 30 people, it is no longer possible to just have the best in class. A large team is necessarily composed of people of different skills, expertise and cost. As in any other industry, the role of the most skilled individuals is not just to be themselves very productive, but more importantly, it is to make the rest of the team productive. This role is typically fulfilled by software architects and development managers. To make their team productive, architects and managers must choose the right tools and practices, not just for the best developers but for the whole team, and they have to make the system immune from individual mistakes and people turnover.

The problem of large teams is not typically one of performance or algorithmic difficulty, but one of scale and complexity that comes from the size of the problem.

Using patterns proved to be a good way to address scale-bound complexity. Large teams typically solve the same problems a large number of times with slightly different inputs (for instance an ERP with 2,000 entities, 25,000 properties and 500 forms). Using a pattern means that all team members will solve the same problem the same way.

The next step for large teams is to use software to help implementing the patterns, or help validating human-written code against the guidelines. Our core product, PostSharp, is exactly designed for this purpose. Other tools can be used: several tools exist to verify compliance of source code against rule, but very few can actually help with the implementation by allowing to express patterns in the programming language.

In this document, we propose to apply this approach to multithreading: to use design patterns (threading models), to deliberately choose productivity over performance, and to use an advanced compiler technology to automate the implementation of the threading model and the validation of source code against the model.

## EXTENDING OBJECT-ORIENTED PROGRAMMING WITH THREADING MODELS

The rest of this paper will describe how we extended the C# and VB languages to support threading models. We did it by using predefined extension points of the language, namely custom attributes ("annotations" in Java terminology).

We did not implement just one threading model. We actually designed an abstraction that allows developers to use several threading models in a single application. So developers are not forced to switch to a different programming language and they don't even need to stick to a single model. They can choose the model that is the most relevant for each part of the application. Threading models are represented as custom attributes that developers can add to each class.

We implemented this approach in the PostSharp Threading Patterns Library 4.0. This is a commercial product (we are one of the few companies that makes a living from selling a compiler extension) and we offer free licenses for evaluation, research and educational purposes. We are convinced that our approach will raise broad interest because it is a fairly general and portable answer to an important problem: bringing thread safety to object-oriented programming.

### Overview of Implemented Threading Models

PostSharp 4.0 implements the following threading model characteristics:

- **Immutable**: an object cannot be modified after its constructor has exited.
- **Freezable**: a relaxation of the Immutable model where an object cannot be modified after the Freeze method has been invoked.
- **Actor:** calls to an object are queued and asynchronously processed by a single thread at a time.
- **Synchronized**: only a single call at a time is allowed on the object; other calls wait before being processed.
- **Reader-Writer Synchronized**: methods are marked as reader or writers; multiple concurrent readers are allowed but writers require exclusivity.
- **Thread Affine**: the object may be invoked only from the thread that instantiated it.

- **Thread Unsafe**: throws an exception if the object is accidentally accessed by two threads concurrently; this is a threading anti-pattern since it does not offer any kind of determinism.

## Object Aggregation and Composition

In object-oriented programming, the unit of reasoning is usually an instance of a class, which is usually called an "object". When we reason about multithreading, we often need to think at a more coarse-grained level than class instance.

Suppose for instance that you need to represent an invoice. Say that an invoice has invoice lines. You will probably need three types to model it: *Invoice*, *InvoiceLine* and *List<InvoiceLine>*. Suppose that the *Invoice* class has a *TotalAmount* property that needs to be set to the sum of the *Amount* property of all lines. You now need to reason about the invoice, including its children classes, as a single entity. If you want the *Invoice* entity to follow the Synchronized model, you need both the invoice and its invoice lines to be protected by the same "lock". If you prefer the Freezable model, it makes sense to recursively freeze the whole object tree rooted at the instance of the *Invoice* class. With the Actor model, we will want to protect access not only to the Actor class itself, but also to all objects forming its state. For all these use cases, it is useful to reason in terms on aggregation and child fields.

Thus, threading models in object-oriented programming are deeply related to the concepts of object aggregation and composition. Aggregation allows to partition the heap into sub-trees that should behave consistently from a concurrency point of view. The result exhibits some similarities to the concept COM apartments, with major differences that will be explained below.

The concepts of aggregation and composition were considered so important to object-oriented design that they were included in the original UML specification. They play a central role in the description of object-oriented design patterns. However, the concepts have not been implemented in mainstream object-oriented languages.

We found it very useful to introduce aggregation back into the programming language.

We define the notion of an Aggregatable class, which is a class whose instances can be involved in a parent-child relationship. Instance fields of an aggregatable class must be annotated with the [*Child*], [*Reference*] or [*Parent*] custom attribute. Omitting to annotate a field will result in a build-time error, unless the field is a value type, a string or a delegate.

The following code snippet shows how an invoice can be represented in C# enhanced by the custom attributes defined by PostSharp 4.0:

```
[Freezable]
public class Invoice
{
    [Child]
    public readonly AdvisableCollection<InvoiceLine> Lines =
           new AdvisableCollection<InvoiceLine>();

    [Reference]
    public Customer Customer;
}

[Freezable]
public class InvoiceLine
{
    [Reference]
    public Product Product;

    [Parent]
    public Invoice ParentInvoice { get; private set; }
}
```

### Advisable Collections

In the above code snippet, notice the use of the *AdvisableCollection* class. This class is defined in PostSharp. It replaces the List and Collection classes. PostSharp also defines *AdvisableDictionary* and *AdvisableKeyedCollection*.

When designing our threading models, we faced the challenge of implementing several kinds of collections: a synchronized collection, a reader-writer synchronized collection, a freezable collection, a thread-affine collection, and so on. Since we also provide an undo/redo feature based on the Aggregatable pattern, we should also have implemented a synchronized recordable collection, a reader-writer synchronized recordable collection and so on. Every one of these collections would be just a plain collection with some additional behavior. You can easily imagine that this design becomes quickly impossible as the number of behaviors and possible combinations increases.

Instead of including the behaviors statically into the collection types, we preferred to develop collections into which we could dynamically add behaviors. If an advisable collection is assigned to a child field of a class, it automatically gets the advices of its parent. In the previous specific example, the collection assigned to the Lines field of the Invoice class is automatically made aggregatable and freezable.

## IMPLEMENTATION STRATEGY

### Software Cell Analogy

Our threading model implementation is designed according to a biological analogy. In biology, every cell is responsible for its own health. Cells have a membrane that separates the inside from the outside. However, no cell lives in isolation; it needs to accept inputs from the outside. Friendly inputs can enter the cell through channels that specifically accept the right kind of molecules. (If you're interested in the biological analogy for object-oriented design, you will find interesting views in Ralf Wesphal's blog.)

Our design is similar. Software cells are modelled as object aggregates. Channels are public and internal methods: they allow the execution flow to enter the object. Access to the object aggregate is *acquired* when the control flow enters a public method and is *released* when it exits the method. Different threading models implement the acquire/release semantic differently. For instance, the Synchronized model uses a monitor and the Thread-Affine model just checks the calling thread.

However, in C# and VB, there are a few scenarios that make it possible to access and even alter the state of an object without first entering a public method:

- Public instance fields.
- Nested classes, including closure classes.
- Delegate calls to private and protected methods.

Forbidding all language features that would allow to break the software cell analogy would not be a viable option. If public instance fields are generally a bad practice and could be forbidden by the compiler, the use of nested and closure classes is legitimate most of the time. It would be theoretically possible to discriminate safe and unsafe use of these language features using abstract interpretation, but this approach is currently not implemented as discussed next.

To work around this issue, we developed two mechanisms. First, we defined the [*EntryPoint*] custom attribute, which marks a private or protected method as a legitimate entry point and requires the compiler to add acquire/release semantics.

Second, we implemented runtime verification of field accesses. When a thread accesses an instance field without first acquiring access to the instance, an exception would be thrown. For instance, accessing a field from a closure class invoked from synchronous LINQ invocation would be allowed but the same closure class could not have access to the field if it were invoked asynchronously from a background thread. In this case, a named (as opposed to anonymous) method should be created and it should be annotated with the [*EntryPoint*] custom attribute.

## Model Checking

Part of the success of Java, compared to C and C++, comes from its ability to guarantee "safe" execution of the code. Safe means that the compiler and the virtual machine guarantee that the code will not read from or write to a memory address of an unexpected type and that execution will not be transferred to something that is not code of the right kind for the current context. Note that the compiler alone cannot guarantee safety. The virtual machine has to do a part of the verification work, for instance by throwing *InvalidCastException* and *NullReferenceException*. The compiler+VM ensemble guarantees that the code succeeds and fails consistently without corrupting the rest of the memory.

There are algorithms to validate that programs are free of invalid casts or null reference accesses. They are based on a technology called abstract interpretation, a branch of static analysis. This is a complex technology involving extensive computational costs, several orders of magnitude more than what simple compilation requires. In the .NET world, Microsoft Code Contracts and Pex are examples of tools implementing concepts from abstract interpretation. Many more tools exist in the Java ecosystem.

Our implementation of threading models follows the same approach.

- **Limited Build-Time Verification**. Because of the time constraints, only simple verification is performed at build time. For instance, we test that a class does not contain public instance fields if the model does not allow for it or that the type of these instance fields has a compatible threading model. None of these verifications involves data flow analysis.
- **Run-Time Verification**. PostSharp generates code that verifies at run-time that the code is used according to the rules set by the threading model. The objective is to provoke deterministic failure as soon as a transgression is detected, even if the transgression would not result in a data race in this particular instance. For instance, it would cause a *ThreadAccessException* in a reader-writer synchronized object when a method tries to write a field without holding write access to the object, even if no other thread is accessing the object. Thanks to extensive runtime verification, a single-threaded test coverage is very likely to detect most defects because threading models tend to make defects more visible and deterministic. Note that run-time verification is enabled only in the debug build configuration by default and is disabled in the release build.
- **Abstract Interpretation** is currently not implemented. One can imagine that abstract implementation could run during the night or even during several days before an important milestone. Most current abstract interpretation tools for multithreading have to validate code that does not follow any threading model. This makes it much more difficult to produce any meaningful analysis. However, abstract interpretation based on threading models could be much efficient and useful because the realm of valid programs is much smaller when validity is tested against a strict model rather than a loose one. There is much potential for development of abstract implementation of model-based multithreaded programs.

PostSharp currently implements exhaustive run-time verification and basic (non-iterative) build-time verification. The objective is not to aim for a complete proof of correctness. Building verifiably correct software is extremely expensive; in the current state of the technology, it only pays off for critical pieces of software such as avionics, OS kernels or hypervisors. Our approach takes the assumption that all software has a reasonable single-threaded code coverage and that most issues will be identified deterministically, i.e. independently from specific thread timings. Defects that are dependent on execution timing are much less likely to affect programs written against threading models because these models are specifically designed to avoid timing-related issues, for instance by forbidding a field to be accessed without a first acquiring a lock. So the models themselves give strong (but not perfect) guarantees, and one can put most energy into testing the code against the model, instead of testing the code for thread safety. We expect that a single-threaded unit test coverage, combined with a reasonable holistic testing, will unveil most model violations. Without threading models, such testing would be unlikely to reveal any threading issue.

## Core Interfaces and Semantics

The previous sections explained how our implementation relied on *acquire access* and *verify access* semantics. In this section, we describe how we translated this strategy into an object-oriented design.

After compilation, all classes that have been assigned a threading model implement the *IThreadAware* interface (see reference documentation). This interface exposes the concurrency controller of the instance.

```
public interface IThreadAware
{
	IConcurrencyController ConcurrencyController { get; }
}
```

Several objects can share the same controller. Typically, child objects share the same controller as their parent.

The concurrency controller has two semantics, exposed by the *IConcurrencyController* interface (see reference documentation): acquiring access to the object aggregate and checking whether the calling thread has access.

```
public interface IConcurrencyController
{
   void Initialize();

   void AcquireAccess( ObjectAccessLevel objectAccessLevel,
                       ref ConcurrentAccessToken concurrencyAccessToken );

   bool CheckAccess( ObjectAccessLevel objectAccessLevel );

   ThreadingModel ThreadingModel { get; }
}
```

The *AcquireAccess* method returns a token that must be disposed of when access is no longer required. The compiler must emit code that invokes *AcquireAccess* before any public or internal method and releases the token when the method exists.

The *CheckAccess* method determines whether the current thread has access to the object aggregate, which is always true if the thread previously invoked *AcquireAccess*. In a checked build, the compiler would emit code that invokes *CheckAccess* and causes an exception if it does not have access.

The *ThreadingModel* property evaluates to a value that represents the threading model implemented by the current concurrency controller.

```
public abstract class ThreadingModel

{
   internal abstract bool RequiresCheckedReads { get; }

   internal abstract bool RequiresCheckedWrites { get; }

   internal abstract bool RequiresAcquire { get; }

   internal abstract bool IsValidChild( ThreadingModel threadingModel );
}
```

The *ThreadingModel* class instructs the compiler whether it should generate code to verify field accesses and access acquisition. The *IsValidChild* method is used at build or run time to verify that children objects are being assigned to a parent of a compatible threading model.

Finally, we have a base custom attribute that allows a threading model to be applied to a class:

```
public abstract class ThreadAwareAttribute
{
    public bool RuntimeVerificationEnabled { get; set; }

    internal abstract IConcurrencyController CreateController();

    internal abstract ThreadingModel ThreadingModel { get; }
}
```

For each threading model, we need to derive three classes:

- the custom attribute itself, derived from *ThreadAwareAttribute*;
- the threading model description, derived from *ThreadingModel*;
- the concurrency controller, derived from *IConcurrencyController*.

## Requirements on the Compiler

So far we have explained the design from syntactic and semantic perspectives. Now we discuss what needs to be implemented by the compiler.

Our threading model implementation relies on object aggregation, so the compiler also needs to implement the aggregatable behavior.

The compiler (or compiler extension) has to add the following logic during compilation:

- When an instance is initialized:
  - Instantiate concurrency controller.
- When the full constructor chain has completed:
  - Initialize concurrency controller (*IConcurrencyController.Initialize*)
- When any field is set (checked build only):
  - Check thread access (*IConcurrencyController.CheckAccess(Read))*
- When a child field is set:
  - Update parent-child relationship and assign child to the parent's concurrency controller.
- When a field is retrieved (checked build only):
  - Check thread access (*IConcurrencyController.CheckAccess(Write))*.
- When a public or internal method is invoked:
  - Acquire and release access (*IConcurrencyController.AcquireAccess)*

## Threading Model Neutrality

Our approach supports several threading models. It does not judge of the quality or appropriateness of any of them. We assume that all threading models have valid use cases in specific situations. In our approach, we assume that it is the responsibility of the architect, and not of the language designer, to select which threading model is relevant for each part of the application.

We do not even assume that an application should have several threads. If the architect decides that a single-threaded application makes sense, all classes can be made thread affine. The benefit over not having any threading model is safety: if some code is accidently invoked from a different thread (for instance an event handler), an exception will be thrown instead of allowing for random data races.

Therefore, the threading framework offers several threading models without judgment of appropriateness. Selection of threading models is a part of the architecture work; the output of this work is typically a set of guidelines and rules, which can be enforced during code reviews or automatically using several tools (just to name a few in .NET: FxCop, Roslyn diagnostics, PostSharp Architecture Framework, NDepend).

## IMPLEMENTATION WITH POSTSHARP

### Understanding PostSharp Technology

The previous sections sketched an object model for thread safety and an implementation strategy. There are different ways to implement this strategy. It would be possible to implement the approach directly in the compiler. Another approach is to use a compiler extensibility technology. We naturally chose the second approach.

PostSharp, our core product, is the leading extensibility technology for C# and VB languages. PostSharp is based on MSIL rewriting. It reads the output of the C# and VB compiler, transforms it and writes it back. PostSharp transparently integrates into the build process so users don't have the impression of working with a post-compiler. They have the feeling of working with an integrated language, the one they already know, enhanced by custom attributes that have real meaning for the build chain.

Although PostSharp finds its inspiration in aspect-oriented programming, its implementation, design and even concepts are completely independent from AspectJ. Aspects in PostSharp are usually (but not necessarily) represented as custom attributes. A major originality of PostSharp is that aspects are *executed* at build time. In PostSharp, an aspect is "something that provides advices."

Aspect providers and advice providers can be expressed programmatically. Typically, an aspect provider would perform some analysis of the current project using the *System.Reflection* API. PostSharp provides a more advanced reflection API for more analysis. For instance, aspect code has access to semantic trees, to object inheritance hierarchies and to declaration usage graphs. This allows it to perform arbitrarily complex analysis and as a result add a set of advices to classes, methods and fields.

An advice in PostSharp is a primitive transformation. Advices are guaranteed to compose safely. PostSharp is intentionally designed to prevent users from performing arbitrary transformations that would break the semantics of the program or make it an invalid program. The level of abstraction of advices is at the same level as the C# or VB language itself. PostSharp developers don't need a deeper understanding of compilers; they must only be proficient C# or VB developers.

Of course, PostSharp has the ability to perform arbitrary MSIL transformations but this functionality is not exposed to end users.

We developed our threading model implementation using PostSharp's high-level APIs, i.e. using means that are accessible, documented and supported for all users.

### Aggregation and Composition

As discussed above, our approach relies on the concept of object aggregation to partition the heap into sub-trees of objects that should behave consistently. All threading models benefit from making a distinction between a child field and a reference field.

Object aggregation is implemented by the Aggregatable aspect (*AggregatableAttribute* class). We designed this aspect independently from the threading models. The Aggregatable aspect is also used by the Disposable aspect (which implements object composition in the UML meaning) and the Recordable aspect (undo/redo feature).

The Aggregatable aspect is responsible for implementing the *IAggregatable* interface, which basically exposes a *Parent* property and a set of children. Children are exposed with a visitor pattern, which can be implemented very efficiently by the compiler.

```
public interface IAggregatable
{
    object Parent { get; }

    event EventHandler ParentChanged;

    event EventHandler<AncestorChangedEventArgs> AncestorChanged;

    RelationshipKind ParentRelationship { get; }

    bool VisitChildren( ChildVisitor visitor, ChildVisitorOptions options );

}
```

The Aggregatable aspect is designed to serve as a base for other aspects. Threading model aspects, but also the Recordable aspect for undo/redo, are "clients" of the Aggregatable aspect. Communication with other aspects is implemented using methods introduced or overridden in the target class of the aspect:

```
public virtual bool VisitChildren( ChildVisitor visitor, ChildVisitorOptions options );

protected virtual void OnAncestorChanged( AncestorChangedEventArgs args )

protected virtual void OnParentChanged();

protected virtual void OnChildAttached( object child, ChildInfo childInfo );

protected virtual void OnChildDetached(object child, ChildInfo childInfo );
```

The Aggregatable aspect typically *introduces* these methods because it is applied first to the target class. The threading aspects, which run after the Aggregatable aspect, then *override* these methods and add adequate behaviors. For instance, the *OnChildAttached* method is used to verify that the child object implements a compatible threading model.

The same mechanism is used by other aspects such as undo/redo. User code can also override these methods to add custom behaviors when a child is attached to a parent.

The Aggregatable aspect is an excellent example of aspect composition that uses the well-understood concept of method overriding as an integration mechanism. The concept of method overriding is simply extended; not only class inheritance can override methods but also aspects.

We believe that the same strategy could be used in other compilers to implement object aggregation and threading models.

Note that PostSharp allows a sub-tree to have its own concurrency boundary. That is, nodes can be synchronization roots even if they are not the absolute root of the tree. By default, an instance is assigned to the concurrency controller of its parent. A class can override this behavior by defining a method named *UseParentConcurrencyController*; if the method returns *false*, the instance will be a synchronization root.

## THREADING MODELS IN DETAILS

### Freezable

The Freezable pattern is a generalization of the Immutable pattern, which is generally inadequate for object-oriented programming. The Immutable pattern requires that an object is not allowed to change after its constructor has completed. However in many object-oriented use cases, initializing an object takes more than just invoking its constructor. A typical initialization would also include initializing fields and properties. Deserialization is a classic example.

The Freezable pattern exists to alleviate the limitations of the Immutable pattern for object-oriented programming. By definition, a freezable object cannot be modified after the *Freeze* method has been invoked.

Our implementation is based on the following principles:

- A freezable object cannot be shared before it has been frozen. If another thread than the one that instantiated it tries to access the object (either in reading or writing), a runtime exception is caused.
- A freezable object cannot be modified after its Freeze method has been invoked.
- The Freeze method also freezes all freezable children.
- Freezable classes can have child fields that are either Freezable or Immutable.

Note that the Freezable model does require the *Acquire* semantic; only the *Check* semantic is applied.

The Freezable aspect introduces the following interface into the target class:

```
public interface IFreezable : IThreadAware
{
    void Freeze();
}
```

### Immutable

The Immutable pattern is implemented as a specific case of the Freezable pattern, where the Freeze method is (conceptually) invoked after the last constructor in the constructor chain is invoked. The immutable aspect does not introduce the *IFreezable* interface.

This implementation strategy contrasts with the definition of immutability that is usually accepted among C# or VB developers. The general belief is that an immutable class is a class whose fields are all read-only. This conception is both excessive and inadequate.

It is excessive because it is valid, from a thread-safety point of view, to modify a field even outside of the constructor as long as the constructor has not fully completed. For instance, it is valid under the immutable model for a constructor to invoke a private method that modifies a field. However, such code would not compile if the field was read-only. Therefore, the constraint of having only read-only fields is excessive.

The conception is also insufficient because it tells nothing about the mutability of objects assigned to fields. If an immutable class has a field of type *List*, it is not enough that the field is read-only; the *List* must also be immutable. In general, all children fields of an immutable class should be an immutable type, or a freezable type if freezable objects are effectively frozen when the constructor exists. Therefore, the read-only keyword is insufficient.

Note that the notion of immutability in object-oriented programming depends on the notion of object aggregation. It is acceptable for an immutable class to contain a field of a non-immutable type if this field represents a *reference* relationship and not a *parental* or *ownership* relationship. Because mainstream object-oriented languages don't have a notion of child or reference, they don't have the ability of implementing immutability properly, which means in a machine-verifiable way.

By building threading models on the top of the concept of object aggregation (which is a largely accepted concept of object-oriented design), it gives the ability to define a well-defined solution to the problem of thread safety.

## Synchronized

The Synchronized threading model is a popular one. It is often implemented by enclosing all public methods by a "lock(this)" statement. It has the effect of allowing a single thread at a time to enter the object.

The model is notorious for causing deadlocks and thread retention. We do not judge of the quality of this model here, but just propose a solution that provides automatic lock acquisition and runtime validation that the lock has indeed been acquired. Note that PostSharp also provides a runtime deadlock detection facility that should make it much easier to diagnose deadlocks. Also, performance issues of this model are only serious if long-running operations (I/O, long computations) are performed when the lock is acquired.

Just as the *lock* keyword, the implementation of the concurrency controller for the Synchronized model is based on the *Monitor* class.

The Synchronized model has both *Acquire* and *Check* semantics.

## Reader-Writer Synchronized

The Reader-Writer Synchronized model is similar to the Synchronized model, but object aggregates are protected by a reader-writer lock instead of a lock.

Public and internal methods must be annotated with one of the following custom attributes: [Reader], [Writer] or [UpgradableReader]. Property getters and setters are automatically considered readers and writers, respectively. Omitting to add a custom attribute to any other public or internal method would result in a build-time warning and a run-time exception.

Note that PostSharp does not attempt to determine whether a method is a reader or writer using static analysis. The reason for this limitation is that it could require a potentially complex analysis in the presence of virtual calls or delegate calls.

The implementation of the concurrency controller currently relies on the *ReaderWriterLockSlim* class.

## Actor

Under the Actor model, any call to a public or internal method is packaged as a message and appended to the actor message queue.

The Actor aspect introduces the *IActor* interface into the target class:

```
public interface IActor : IThreadAware
{
    IActorDispatcher Dispatcher { get; }
}
```

The *IActorDispatcher* interface is the implementation of the message queue. The default implementation is based on the *ConcurrentQueue* class. Processing of the queue is done from the .NET thread pool. Developers can provide their own implementation of the dispatcher by implementing the *IActorDispatcher* interface manually in the target class. For instance, it is possible to create a dispatcher whose execution is affined to a specific CPU core or that uses a different data structure than a *ConcurrentQueue*.

The Actor pattern requires all methods either to have no return value or to be asynchronous (*async* keyword in C#). Violating this rule would result in a build-time error.

The Actor concurrency controller does not have an *Acquire* semantic. Instead, public methods are made asynchronous. For methods that are already asynchronous, the state machine is lightly modified to make sure that it executes in the context of the actor. This transformation is the only one that relies on the low-level APIs of PostSharp SDK, because the public APIs cannot fulfill the requirements.

Thanks to this transformation, async methods are always asynchronous. In the vanilla C# async state machine, the first chunk of the method (before the first *await* statement) runs synchronously.

## Thread Affine

Both WinForms and WPF follow the Thread Affine model. Our implementation simply remembers the thread that created the instance and verifies the current thread both in *Acquire* and *Verify* semantics of the concurrency controller.

## Thread Unsafe

The Thread Unsafe model is obviously not a recommended threading model. It exists to ease diagnosis of multithreading issues. Sometimes data corruption happens in an object that is not supposed to be accessed concurrently by several threads.  Instead of allowing for data corruption, the Thread Unsafe aspect will in this case cause an exception of type *ConcurrentAccessException.*

The *Acquire* semantic of the concurrency controller simply stores the current thread in a field and causes an exception if the field was already assigned to a different thread. Of course, the verification is implemented using an interlocked *CompareExchange* operation.

The concurrency also implements the *Check* semantic. If a field is accessed from a thread that didn't acquire access to the object, an exception would be caused. This makes sure that it actually captures all concurrent accesses.

This aspect does not provide any kind of guarantee but can ease the diagnosis of existing issues by causing an early failure instead of data corruption.

## Threading Model Table

The following table describes how each threading model differently implements the Acquire semantics of the concurrency controller.

| Model | Acquire | Verify |
|---|---|---|
| **Freezable** | Before Freeze:<br><br>Throws ThreadMismatchException is access is required from a different thread than the constructor thread<br>(i.e. equivalent to Thread Affine)<br><br>After Freeze:<br><br>Throws ObjectReadOnlyException if write access is required. | Same as Acquire |
| **Immutable** | Same as Freezable, but the object is considered frozen after the last constructor is invoked. | Same as Acquire |
| **Synchronized** | Conceptually equivalent to Monitor.Enter(this.ConcurrencyController) | Throws ThreadAcccessException if the current thread does not (conceptually) own the lock on this.ConcurrencyController |
| **ReaderWriterSynchronized** | Acquires a ReaderWriterLockSlim in this.ConcurrencyController with the right level of access. | Throws ThreadAcccessException if the current thread does not own access to ReaderWriterLockSlim. |
| **Actor** | Not Applicable (public methods are made asynchronous and dispatched to the proper context) | Throws ThreadMismatchException if accessed from invalid thread. |
| **Thread Affine** | Throws ThreadMismatchException is access is required from different thread than constructor | Same as Acquire |
| **Thread Unsafe** | Same than Synchronized, but does not wait for the monitor, and throws ConcurrentAccessException if monitor cannot be acquired without waiting. | Same as Synchronized |

# COMPOSABILITY AND COMPATIBILITY

This chapter discuss how threading models can be composed with each other and with other concurrency concepts of C# and VB.

## Mutual composability

### Composability through aggregation

When reasoning about aggregation of thread-aware objects, we need to address the question of compatibility of threading models with each other.

For instance, it is clear that immutable classes are valid for child fields of an immutable class, but what about a freezable or synchronized class?

The criteria we used to determine composability is the following: does the child threading model preserve the characteristics (guarantees) of the parent threading model? The idea is that we always reason on object trees, so all assumptions that are true for the parent must be also true for all children.

A second question is whether the child object will share the same concurrency controller as the parent. Clearly, immutable, freezable or thread-affine objects don't need to share the same concurrency controller as their parent. The *IConcurrencyController* interface is generally implemented by the object itself, and its implementation is trivial. Other threading models (actors and lock-based models) must share the same threading model as their parent.

When the concurrency controller of a child is being replaced by the one of its parent, the threading model needs to remain consistent. The client code of a class should not have to consider the threading model of the parent, especially since the parent may be unknown at build time. Therefore, the parent threading model should also preserve the guarantees of the child threading model.

Note that threading model guarantees affect only access to the mutable state. This gives the *immutable* and *freezable* models a highly-compatible status because there is no mutable state.

The assumptions are the following for each threading model:

- **Freezable**, **Immutable**: no write access allowed after freezing or construction.
- **Actor**: access to mutable state only from the proper context, no thread affinity over the whole actor lifetime.
- **Synchronized**: concurrent access (from the point of view of the caller) to mutable state is allowed but may result of the current thread to wait.
- **Reader-Writer Synchronized**: same as Synchronized, but waiting is less likely.
- **Thread Affine**: mutable state accessed only from a single thread, mutable state is guaranteed to be modified only by the current method.
- **Thread Unsafe**: no guarantee.

The following table describes the compatibility of threading models along the lines of parent-child relationships:

| **Row: Parent Column: Child** | **Actor** | **Freezable** | **Immutable** | **Reader-Writer Synchronized** | **Synchronized** | **Thread Affine** | **Thread Unsafe** |
|---|---|---|---|---|---|---|---|
| **Shared Controller** | **Yes** | **No** | **No** | **Yes** | **Yes** | **No** | **Yes** |
| **Actor** | Yes* | Yes | Yes | Yes* | Yes* | No | Yes |
| **Freezable** | No | Yes | Yes | No | No | No | No |
| **Immutable** | No | Yes | Yes | No | No | No | No |
| **Reader-Writer Synchronized** | No | Yes | Yes | Yes | No | No | Yes |
| **Synchronized** | No | Yes | Yes | Yes | Yes | No | Yes |
| **Thread Affine** | No | Yes | Yes | No | No | Yes | Yes |
| **Thread Unsafe** | Yes | Yes | Yes | No | No | Yes | Yes |

* Although the pair would be conceptually compatible, it is not currently compatible in our implementation.

### Composability through parameter passing

Our current design does not set any constraint on the threading model of parameters.

In theory, a threading language extension should require that the parameters of thread-crossing methods are thread-safe. Some threading models may have stricter requirements.

For instance, the classic Actor model requires all arguments to be immutable. Our design does not implement this restriction, although developers could implement the constraints themselves using aspects.

## Asynchronous Methods

The software cell analogy makes perfect sense for conventional synchronous methods. However, coping with co-routines require more discussion. C# and VB have two kinds of co-routines: iterators and asynchronous methods. We will limit the scope of this discussion to asynchronous methods.

The particularity of asynchronous methods is that their execution can be suspended and resumed. The *Suspend* and *Resume* semantics have different interpretations according to the threading model of the declaring class. However, in all threading models, the *Suspend* semantic introduces the possibility of reentrancy.

Under the *thread affine* model (such as UI objects), asynchronous methods have no other meaning than to introduce reentrancy at well-known points of execution. By suppressing reentrancy, one would also force the affined thread to wait for completion of the awaited task, which would basically turn the asynchronous method into a synchronous method. Note that the Windows operating system allows for some level of reentrancy even in synchronous code, by allowing the message queue to be processed while a method is waiting for thread synchronization objects. This design decision minimizes the possibility of deadlocks.

The reentrancy-versus-deadlock dilemma is present with other models as well.

Under the *actor* model, one has to decide whether the actor is allowed to process other method calls while another method call is being suspended, or whether processing of the message queue is also suspended. Allowing other method calls makes the actor reentrant; forbidding them makes it prone to deadlocks.

Under the *synchronized* or *reader-writer synchronized*, the dilemma is to decide whether access exclusivity (or "lock") should be held or released when a method call is suspended. If the access is held, we introduce a possibility of deadlock. If access is released and then acquired back, the state of the object is not guaranteed to be preserved across the awaiting point. That is, we introduced a form or reentrancy to the synchronized object.

Under the *freezable* and *immutable* models, the discussion of reentrancy is irrelevant because the state cannot change.

For all other models, it is important to answer the question of reentrancy. We believe that this question should be answered by the developer and not by the language designer.

Since the aim of our design is to maximize code safety for average developers, we take the assumption that deadlocks are better than reentrancy because deadlocks are more deterministic than data corruption due to reentrancy. Deadlocks are traditionally difficult to diagnose, but this is due to of insufficiencies of application frameworks. Deadlock detection algorithms are well-understood and routinely implemented in database engines. Their use could be generalized to application frameworks. PostSharp implements a deadlock detection facility that instruments most synchronization primitives of the .NET Framework, as well as the concurrency controllers of the threading models.

Therefore, we decided that reentrancy should be an opt-in feature.

In theory, reentrancy is a characteristic of the suspend point. In .NET languages, it could be represented as a qualifier of the *await* statement. This would not be the first qualifier of the *await* statement; the method *Task.ConfigureAwai*t of the .NET Framework fulfills the same goal. We could use a similar mechanism and define a *Task.Reentrant* extension method.

In our current implementation, do not allow to tag individual await points as reentrant, but we ask the whole method to be marked with a *[Reentrant]* custom attribute. The attribute means that all await points in the asynchronous method are safely reentrant.

## Iterators

Although iterators (*yield return* in C#) are also co-routines and in theory may be processed identically to asynchronous methods, we assumed that waiting and reentrancy was not the primary goal of this construct, so we apply the blocking behavior by default. For instance, if an iterator of a synchronized method would hold access during until it is disposed.

## Other existing multi-threading primitives

Object-oriented languages typically have some support for multi-threading. In the case of C# and VB, these primitives are:

- Volatile fields;
- Lock keyword;
- Classes libraries;

- Asynchronous methods (async/await).

Except asynchronous methods (discussed above), threading models are not *generally* composable with other multithreading primitives. The principal reason is that threading models operate at a higher level of abstraction. There should be no need to use volatile fields, locks or other primitives from the class library, as it would defeat the purpose of using threading models.

However, just as the C language allows inline assembly code, it is possible to use any primitive from a class that has been assigned to a threading model.

In our implementation, developers can opt-out from the validation and transformations added by threaded models by adding the *[ExplicitlySynchronized]* custom attribute on a field or a method.

## BENEFITS

Independently from implementation-specific details, the proposed approach has several benefits compared to current practices in mainstream software development.

### Multiple side-by-side models

Our approach enables a single application to use several threading models side by side. In most other approaches, one or two threading models are built inside the programming language, and moving to a different model requires the code to be ported to a different language or platform.

### More concise source code

Because the most thread-synchronization logic is generated by the compiler, it takes less source code to implement the same feature set. Since the total cost of building software is in a super-linear relationship to its number of lines of code[1], this results in important cost saving both during development and maintenance.

### Simpler Source Code

Addressing multi-threading at the model level relieves developer from the need to write code at a low level of abstraction. Many technical details are outsourced to the compiler. Threading models work as a contract between the developer and the compiler. When a class is marked as immutable, the compiler will ensure it actually is, and will raise build- and run-time errors in case of model violation. Developers can cognitively focus on the fact that the fact is immutable; they don't have to worry about the implementation that actually makes it immutable.

In short, threading model makes it simpler to write reliable multithreaded software. This means that writing and maintaining the code requires less qualifications, and that the most qualified developers can spend their time in activities of higher value. This is important in an industry that is characterized by scarcity of talent.

### Fewer Defects

As discussed above, programming against threading models allows the code to be automatically verified against the model. Therefore, most defects can be detected at an earlier stage than before, either at build

[1] This is a statistically established fact. For details, see COCOMO or Software Estimation.

time or during single-threaded unit tests. Non-deterministic defects are much less likely than in code that does not follow a threading model.

### Named Design Patterns

Threading models are a special category of design patterns, so they exhibit the same benefits. Design patterns largely benefit from being a standard part of the body of knowledge of the profession of software development. Design patterns have a name that is recognized by everybody in the profession. So, when someone says “these objects form a chain of responsibility”, all team members understand what is meant. Design patterns form concepts that developer use to reason. These concepts have a name that are used to communicate. Therefore, design patterns extend the natural language of the profession, just as other realms have words to design wholes instead of referring to their parts and their specific configuration. Mature professions typically have an elaborate lexicon.

Programming languages allow people to build a machine-executable executable representation of a problem, that is, programming languages are a human-machine interface. A perfect programming language would exactly match the level of abstraction of the natural language. Object-oriented programming has named types, named methods, named fields, but artefacts like patterns cannot be represented in conventional programming languages. Therefore, patterns are used as reasoning and communication tools during design, but they are lost in source code. Conventional programming language therefore produce an abstraction gap between the design and the implementation.

Our approach to threading models explicitly enriches the programming language with terms that describe the realm of multithreaded programming, and reduces the abstraction gap.

### Ease of implementation

The approach defines a general framework of which individual threading models are specific instances. A large amount of compiler logic can be reused among threading models. Some models, like Actor and Transacted, require specific compiler logic.

## LIMITATIONS

### Limited Build-Time Model Checking

PostSharp only performs structural model checking, not behavioral checking. Complete model checking could be implemented using abstract interpretation. The application of this technology to the checking of code against the threading models presented here could be much more successful than other threading verification methods because with threading models, the verifier actually understands the threading semantics of the code at a high level. Therefore, there is a large potential in applying abstract interpretation to the verification of threading models.

Several kinds of build-time verification are possible:

- The “software cell” analogy is respected (no access to an instance without going through a public/internal method).
- Thread affinity is respected (no call to a GUI-bound object from a non-GUI thread).
- Arguments passed to a thread-crossing method are thread safe.
- Static fields are all read-only and of a thread-safe type.

### Blocking waits in async methods

Our implementation of the synchronized and reader-writer synchronized threading models is currently based on locks. This means that the concurrency controller would wait synchronously for the lock instead of awaiting for it asynchronously. Support for asynchronous awaiting would require improvements in the ability of PostSharp Aspect Framework to modify state machines.

### Only Reentrant Asynchronous Methods

Our current implementation requires all asynchronous methods to be annotated with the *[Reentrant]* custom attribute, otherwise a build-time error is emitted.

### No Asynchronous Waiting

When re-acquiring the access after *await*, an asynchronous method would wait synchronously for the access to be available. A proper implementation should await without blocking the current thread.

### No Host Protection

The current implementation has not been test for robustness under non-deterministic exceptions like *OutOfMemoryException* or *ThreadAbortException*.

## KEY IDEAS

The proposed approach relies on several ideas whose are not all original but whose combination, we believe, is worth attention:

- The focus on cognitive aspects of programming and a vision of computing history that takes cognitive aspects into account.
- The idea that notions like immutable, freezable, actor, locking, can be unified under a concept of “threading model”.
- The idea that threading models apply to classes instead of a whole application or component, or to fields or blocks of code.
- The idea that many models can coexist in the same application.
- The idea that programming languages can be extended using language extensions. In contrast, mainstream beliefs affirm that class libraries (“frameworks”) are the only extensibility points of the language, and that one must move to a different language to get a language-integrated threading model.
- The idea that design patterns can be expressed in the language using language extensions. In contrast, the mainstream practice which is to generate boilerplate code from design patterns, then to enforce compliance by code review.
- The idea to restrict the definition of what is valid source code, on a per-class basis, to enforce a subset of C# that can be verified with a reasonable amount of precision. The same idea was applied to the memory model before. In contrast, the mainstream practice allows any construct in all situations even if it leads to code that is hard to reason about (exceptions to this rule, such as *unsafe* regions in C# or FxCop rules, show that this idea is a good one and it’s worth extending its use).
- The use of Aggregation/Composition as a foundation and the formalization of composability of threading models along the lines of parent-child relationship (e.g. freezable can be a child of an immutable, thread-affine cannot), so that the heap can be partitioned in different “apartments”.

- An object model of threading models that make their implementation relatively easy by the compiler, the runtime engine and the class library. For instance, it allows to create a single "thread-aware collection" instead of an "immutable collection", "freezable collection", "synchronized collection", …

## SUMMARY

Object-oriented programming is the result of decades of improvements of two programming models: execution and memory. Today, the challenge of our generation is to address the problem of multi-threaded programming. Although many have called for abandoning object-oriented programming in favor of functional programming, we believe this cry is not only premature but inappropriate. If we understand what has made object-oriented successful, we give ourselves the ability to find an equally successful solution to the problem of multithreading.

In this paper, we proposed an abstraction and an object-oriented design for threading models. We explained the role of a few interfaces, principally *IThreadAware* and *IConcurrencyController.*

Like many design patterns, threading models rely on the notion of object aggregation. This concept is central to object-oriented design (it is a part of the UML core specification), but has not made it to mainstream languages. To solve the problem of multi-threaded object-oriented programming, we first need to bring back aggregation to the language.

We designed an extension to C# and VB that relies on standard extensibility points of these languages: custom attributes. Aggregation is simply represented by the [Child], [Reference] and [Parent] field-level custom attributes. Threading models are represented as class-level custom attributes: [Synchronized], [ReaderWriterSynchronized], [Actor], [ThreadAffine], [Freezable] or [Immutable].

These interfaces and customer attributes define the syntax and semantics of our language extension. There are several options to implement the language extension and we described our implementation using PostSharp. The tool acts like a compiler extensibility technology for C# and VB and combines several technologies including an aspect framework freely inspired from aspect-oriented programming and static analysis features.

The approach we propose does not allow for complete soundness verification at build time but instead fast build-time structural verification with fast run-time behavioral checking. As a result, failures are made as deterministic as possible and a single-threaded test coverage is likely to expose most defects. This approach is actually very similar to the way that Java and C# enforce type safety. More advanced model checking could be achieved later using abstract interpretation.

We believe that this pragmatic approach may greatly improve the productivity of mainstream developers building multithreaded business, web and mobile applications. Developers gained productivity when moving from assembly to C and then from C to C#. This is because they adopted more abstract execution and memory models and indeed lost control over irrelevant technical details of their code. We believe that the same productivity gain can be achieved by raising the level of abstraction at which multithreading is addressed.

## FURTHER RESOURCES

Download PostSharp: http://www.postsharp.net/download

Request free PostSharp license (MVPs, students, teachers, freelancers): http://www.postsharp.net/purchase/request-free

Online Documentation: http://doc.postsharp.net/threading-library