Project Zonnon:
The Language, The Compiler,
The Environment

Eugene Zouev

Bergen Language Design Laboratory
Bergen University
May 19, 2010
Outline

Project History
Zonnon Language
Zonnon Compiler
CCI & Zonnon Compilation Model
Integration into Visual Studio
Zonnon Builder
Link, Conclusion, Acknowledgements
Project History

1999, Oberon.NET
Projects 7 & 7+ launched by Microsoft Research

2001, Active Oberon
ETH Zürich; the notion of active object

2004, Active C#
ETH Zürich; communication mechanism based on syntax-oriented protocols

2004-2006, Zonnon
Zonnon Highlights

A member of the family of Pascal, Modula-2, Oberon: compact, easy to learn and use

Supports modularity (with importing units and exporting unit members)

Supports object-oriented approach based on definition/implementation paradigm and refinement of definitions

Supports concurrency based on the notion of active objects and syntax-based communication protocols
Zonnon Program Architecture 1

Module

Definition

Object

Implementation
Zonnon Program Architecture 1

Module
- System managed object:
  - encapsulates resources
  - implements definitions
  - aggregates implementations

Definition
- Unified unit of abstraction:
  - represents abstract interface
  - refines another definition

Object
- Program managed actor/resource:
  - implements definitions
  - aggregates implementations
  - specifies concurrent behaviour

Implementation
- def.implementation of a definition
- standalone aggregation unit
Zonnon Program Architecture 2

Definition D1

Definition D11 (refines D1)

Implementation D11 (default for Def D11)

Object A (implements D1 & D11 without reusing def implem)
Zonnon Program Architecture 2

**Definition D1**

**Definition D11**
(refines D1)

**Definition D2**

**Definition D3**

**Implementation D3**
/default for Def D3/

**Implementation D11**
/default for Def D11/

**Object A**
/implements D1 & D11 without reusing def implem/

**Object B**
/implements D2, reuses default implem for D3, and aggregates I/

**Implementation I**
/standalone resource/
Definitions

(* Common interface for all kind of vehicles *)
definition Vehicle;
  var { get } Speed : integer; (* read-only *)
  procedure SpeedUp ( d:integer );
  procedure SlowDown ( d:integer );
end Vehicle.

definition Truck refines Vehicle;
  (* Inherits interface from Vehicle *)
  const SpeedLimit = 90;
end Truck.
Definition & Implementation

(* Common interface for random numbers generators *)
definition Random;
  var { get } Next : integer;  (* read-only *)
  procedure Flush; (* reset *)
end Random.

(* A default implementation of the generator *)
implementation Random;
  var { private } z : real;
  procedure { public, get } Next : integer;
    const a = 16807; m = 2147483647; q = m \ div \ a; r = m \ mod \ a;
    var g : integer;
    begin g := a*(z \ mod \ q) - r*(z \ div \ q);
        if g>0 then z := g else z := g+m end;
    return z*(1.0/m)
  end Next;
  procedure Flush; begin z := 3.1459 end Flush;
begin Flush
end Random.
Definitions & Objects

(* Common interface for the random numbers generator *)
definition Random;
    var { get } Next : integer; (* read-only *)
    procedure Flush; (* reset *)
end Random.

(* A custom implementation of the generator *)
object myRandom implements Random;
    (* Procedure Next is reused from default implementation *)
    (* Procedure Flush is customized *)
    procedure Flush implements Random.Flush;
    begin
        z := 2.7189
    end Flush;
begin
    Flush
end myRandom.
module Test;
  import Random, (* both definition and implem are imported *)
  myRandom;
  var x : object { Random };
    (* x's actual type is either Random or any type
      implementing Random *)

  object R2 implements Random;
    (* Another implementation of Random definition *)
    ...
    end R2;
begin
  x := new Random;
    ...
  x := new myRandom;
    ...
  x := new R2;
end Test.
System-wide activity is scheduled by evaluating the set of all AWAIT preconditions.
Activity Example: A Pipeline with Active Objects

Object Stage (next: Stage);
  var { private } n, in, out: integer;
  buf: array N of object; (*used as a circular buffer*)

procedure { private } Get (var x: object);
begin { locked } (*mutual exclusion*)
  await (n # 0); (* precondition to continue*)
  dec(n); x := buf[out]; out := (out+1) mod N
end Get;

procedure { public } Put (x: object);
begin { locked } await (n # N); (*mutual exclusion*)
  inc(n); buf[in] := x; in := (in+1) mod N
end Put;

Activity Processing; var x: OBJECT;
begin loop Get(x); (*process x;*) next.Put(x) end
end Processing;

begin (*initialise this new object instance*)
  n := 0; in := 0; out := 0; new Processing;
end Stage;

Wait whilst the buffer is empty
Wait whilst the buffer is full

Each activity has a separate thread

Taken from a talk of JG, BK, and DL
Activities & Protocols

\textbf{definition} \texttt{D};
\begin{verbatim}
  \textbf{protocol} \texttt{P} = (a, b, c); (* declaration of a protocol *)
\end{verbatim}
\textbf{end} \texttt{D}.

\textbf{object} \texttt{O}; \textbf{import} \texttt{D};
\begin{verbatim}
  \textbf{activity} \texttt{A} \textbf{implements} \texttt{D.P}; (* declaration of an activity *)
  \begin{verbatim}
    begin ... return u, v, w; (* activity returns tokens *)
      x, y := \textbf{await}; (* activity receives tokens *)
    \end{verbatim}
  \textbf{end} \texttt{A};
  \textbf{var} \texttt{p} : \texttt{P}; (* declaration of an activity variable *)
  \begin{verbatim}
    begin
      p := \texttt{new} \texttt{A}; (* create an activity *)
      (* Continued dialog between caller and callee *)
      p(x, y); (* caller sends tokens x, y to activity p*)
      u, v, w := \textbf{await} \texttt{p}; (* caller receives tokens from p *)
      if \texttt{u} = \texttt{P}.a then ... \textbf{end}; (* using the token received from p *)
      r := p(s, t); (* same as a(s, t); r := \textbf{await} \texttt{p} *)
      \textbf{await} \texttt{p}; (* wait for activity to terminate *)
    \end{verbatim}
\textbf{end} \texttt{O}.  
\end{verbatim}
Syntax-Based Protocols

definition Fighter;
  (* See full example in the Zonnon Language Report *)
  (* The protocol is used to create Fighter.Karate activities *)
protocol (* syntax of the dialog*)
  { fight = { attack ( { defense attack } |
      RUNAWAY [ ?CHASE ] |
      KO | fight ) }.
    attack = ATTACK strike.
    defense = DEFENSE strike.
    strike = bodypart [ strength ].
    bodypart = LEG | NECK | HEAD.
    strength = integer. }
  (*enumeration of the dialog elements to be exchanged*)
Karate = ( RUNAWAY, CHASE, KO, ATTACK, DEFENSE, LEG, NECK, HEAD );
end Fighter.
Outline

Project History
Zonnon Language
Zonnon Compiler
CCI & Zonnon Compilation Model
Integration into Visual Studio
Zonnon Builder
Link, Conclusion, Acknowledgements
Zonnon Compiler

Compiler front-end is written in C# using conventional compilation techniques (recursive descent parser with full semantic control).

Compiler uses CCI framework as a code generation utility and integration platform.

Three versions of the compiler are implemented (all share the single core):
- command-line compiler
- compiler integrated into Visual Studio
- compiler integrated into Zonnon Builder
Just Demo:
Binary Search
Outline

Project History
Zonnon Language
Zonnon Compiler
CCI & Zonnon Compilation Model
Integration into Visual Studio
Zonnon Builder
Link, Conclusion, Acknowledgements
Common Compiler Infrastructure

Universal framework for developing compilers for .NET and integrating them into Visual Studio

Supports CLR-oriented semantic analysis, program tree building and transformation, name resolution, error processing and IL+MD generation; doesn’t support lexical & syntax analyses

Can also be used as a faster alternative to System.Reflection library

 Doesn’t require COM programming: C# only

Implemented in Microsoft; is used in Cω & Spec# compilers (as well as in their predecessors)
CCI Architecture

Compiler Front End

Semantic Representation

Integration Service

Compiler Back End

Assembly Generation Service

Visual Studio .NET
CCI Major Parts

Intermediate Representation (IR) -
A rich hierarchy of C# classes representing most common and typical notions of modern programming languages.

Transformers (“Visitors”) -
A set of classes performing consecutive transformations \( \text{IR} \Rightarrow \text{MSIL} \)

Integration Service -
Variety of classes and methods providing integration to Visual Studio environment (additional functionality required for editing, debugging, background compilation, project management etc.)
CCI Way of Use: Common Principles

All CCI services are represented as classes. In order to make use of them the compiler writer should define classes derived from CCI ones. (The same approach is taken for Scanner, Parser, IR, Transformers, and for Integration Service)

Derived classes should implement some abstract or virtual methods declared in the base classes (they compose a “unified interface” with the environment)

Derived classes may (and typically do) implement some language-specific functionality.
Prototype parser: abstract class from CCI

Parser’s “unified interface”: implementation of the interface between Z compiler and environment

Z parser’s own logic
CCI Compilation Model 1

Source Language Part:
- Language specific

CCI Part:
- Common to all languages

- Source
- IR (AST)
- Scanner & Parser
- Visitors
- MSIL+MD
- IL/MD Reader
- Imported Assemblies
- IL/MD Writer
- Output Assembly
CCI Compilation Model 2

CCI IR Hierarchy

CCI Base Transformers

Visitor 1

Visitor 2

Visitor N

MSIL+MD
CCI Compilation Model 2

CCI IR Hierarchy

CCI Base Transformers

Visitor 1

Visitor 2

Visitor N

MSIL+MD

X Language IR Hierarchy

X Language Transformers

Visitor 1

Visitor 2

Visitor 3+

Visitor M

MSIL+MD
CCI Compilation Model 3: Example
Extending the IR Hierarchy

Ada `exit` statement: `exit when <Condition>;`
CCI Compilation Model 3: Example
Extending the IR Hierarchy

1. Extend existing Exit node

   public class AdaExit : Exit {
     Expression condition;
   }

2. Add new statement to the hierarchy

   public class AdaExit : Statement {
     Expression condition;
   }

3. Use semantic equivalent from the existing hierarchy

   Represent Exit as an instance of class If with condition == <Condition>, falseBlock == null, and trueBlock == Block with one element of type Exit.
Extending a Visitor

using System.Compiler;

namespace AdaLanguageCompiler
{
    public sealed class Looker : System.Compiler.Looker
    {
        public override Node Visit(Node node)
        {
            switch (node.NodeType)
            {
                case NodeType.AdaExit:
                    return this.VisitAdaExit(node);
                default:
                    return base.Visit(node);
            }
        }

        private If VisitAdaExit(AdaExit node)
        {
            /* Transform AdaExit node to If node */
        }
    }
}
Zonnon Compilation Model 1

Zonnon IR Hierarchy

(A Subset of) CCI IR Hierarchy

Zonnon Transformers

(A Subset of) CCI Base Transformers

Visitor K

Visitor N

MSIL+MD
Zonnon Compilation Model 2
Example: Zonnon Tree & Transformers

```csharp
public sealed class DEFINITION_DECL : UNIT_DECL
{
    // Constructor
    public DEFINITION_DECL ( Identifier name ) : base(name) { }
    // Structure
    public DECLARATION_LIST locals; // members, procedure headings
    public UNIT_DECL base_definition;
    public UNIT_DECL default_implementation;

    // Fills the structure after parsing the source
    public static DEFINITION_DECL create ( IDENT_LIST name, MODIFIERS modifiers ) { ... }
    // Resolves forward declarations
    public override NODE resolve ( ) { ... }
    // Checks semantic correctness
    public override bool validate ( ) { ... }
    // Generates CCI node(s)
    public override Node convert ( ) { ... }
}
```

convert transformer encapsulates mappings Zonnon->CLR
Zonnon Compilation Model 3

Why two trees:

- Reflect the conceptual gap between Zonnon and the CLR
- Zonnon semantic representation is kept independent from the CCI and the target platform
- Conversion Zonnon tree -> CCI tree explicitly implements and encapsulates mappings from the Zonnon language model to the CLR
Some Mappings Zonnon->CLR

**definition D:**

```ml
var x : T;
const k = 10;
type e = (a,b,c);
procedure p ( y : T );
end d.
```

**interface D:**

```csharp
interface D {
    T x { get; set; }
    // Nothing
    // Nothing
    void p ( T y );
}
```

**implementation D:**

```csharp
public class D_implem : D {
    enum e { a, b, c }
    T x, y;   // x hides D's x
    void p ( T y )
    {
        ...x...   // “native” x
        ...k...   // D_default.k
        ...e...   // D_default.e
    }
}
```
Zonnon Compilation Model 5

Some Mappings Zonnon->CLR

```plaintext
object O implements d;
    procedure p ( y : T ) implements d.p;
    begin
        ...x...
    end p;
end O.

public sealed class O : D, D_implem // Option 1
{
    public override void p ( T y ) { // hides D_implem's p()
        ...x...
    }
}

public sealed class O : D // Option 2
{
    private D_implem mixed;
    public override void p ( T y ) {
        ...mixed.x...
    }
}
```
Outline

Project History
Zonnon Language
Zonnon Compiler
CCI & Zonnon Compilation Model
Integration into Visual Studio
Zonnon Builder
Link, Conclusion, Acknowledgements
Compiler Integration: Traditional Approach

Source File Name
Compilation Params

Compiler Start Up

Source Code
Lexical Analysis
Sequence of Tokens
Syntax & Semantic Analysis
Program Tree
Code Generation
Object Code

Compiler End Up

Compiler as a “Black Box” Program

File with Object Code
Diagnostic Messages

Environment
Compiler
What Does Integration Assume? 1

Visual Studio Components

- Project Manager
- Text Editor
- Semantic Support ("Intellisense")
- Debugger

Features That Should be Supported by a Compiler

- Language sources identification
- Syntax Highlighting
- Automatic text formatting
- Smart text browsing { ➔ }
- Error checking while typing
- Tooltip-like diagnostics & info
- Outlining (collapsing parts of the source)
- Type member lists for classes and variables of class types
- Lists of overloaded methods
- Lists of method parameters
- Expression evaluation
- Conditional breakpoints
What Does Integration Assume? 2

Example of “Intellisense” Feature
Compiler Integration: CCI Approach

Compiler as a Collection of Resources
Compiler Integration: CCI Approach

---

- **Document**
  - Source Code
  - Source Context

- **Token**
  - Token Attributes
  - Token Context

- **Program Tree**

- **Object Code**
  - (Assembly)

---

- **Compiler as a Set of Objects**

---

- **Environment**
  - Source Text Editor
  - Project Manager
  - “Intellisense” etc
  - Debugger

---

- **Lexical Analysis**
- **Syntax & Semantic Analysis**
- **Code Generation**
Outline

Project History
Zonnon Language
Zonnon Compiler
CCI & Zonnon Compilation Model
Integration into Visual Studio
Zonnon Builder
Link, Conclusion, Acknowledgements
Zonnon Builder

A standalone, easy-to-use integrated development environment: convenient for beginners and looks familiar to Pascal programmers

A simple and light-weight alternative to Visual Studio

Supports a typical development cycle comprising source code editing, compiling, execution, testing, debugging, project management, file versioning

Supports a simplified development cycle where a single program file is being developed, compiled, debugged and run
Zonnon Builder

Just Demo: Chess Notebook Program
Zonnon Web Page

www.zonnon.ethz.ch

Zonnon program samples
(including Chess Notebook),
Zonnon Test Suite (1000+ test cases),
Zonnon Language Report,
Related Papers and Talk Slides,
Zonnon Compiler Distribution
(updated almost every Monday)
Conclusion

Zonnon is a new programming language which combines conventional notation and classic modularity with modern and powerful paradigms like object orientation and language-level concurrency.

Zonnon can be used together with other .NET languages within the same environment (Visual Studio).

To the best of our knowledge, the Zonnon compiler is the first compiler developed outside of Microsoft that is fully integrated into Visual Studio.

Zonnon is used for teaching minor students programming (as the first language) in Nizhny Novgorod university, Russia.
People Involved

J. Gutknecht, ETH Zürich
Primary Language Author

B. Kirk, Robinson Associates

D. Lightfoot, Oxford Brookes University
Zonnon Language Report

H. Venter, Microsoft
Common Compiler Infrastructure

E. Zouev, ETH Zürich
Zonnon Compiler, Integration into VS

V. Romanov, Moscow State University
Zonnon Test Suite, Zonnon Builder, Chess NB

A. Freed, NASA
First “Industrial” Zonnon User

V. Gergel, R. Mitin, NN State University, Russia
An Introductory Course in Programming based on Zonnon; Zonnon Program Samples
Questions?
Suggestions?
Critique?