

Zonnon for .NET

A Language and Compiler Experiment

Jürg Gutknecht, Eugene Zueff

Computer Systems Institute, ETH Zürich, Switzerland
{gutknecht,zueff}@inf.ethz.ch

Abstract. In this article we present the principles of design of a new programming language called Zonnon and its implementation for .NET. Zonnon is an evolution of Oberon, distinguishing itself by a wide spectrum of applications and by a highly uniform and comprehensive object model. Special highlights are an advanced notion of active object and a unified concept of abstraction called definition. We briefly discuss both the mapping of Zonnon's object model to .NET and the use of a new compiler integration technology called CCI.

1. Introduction

This is a report on work in progress. The project emerged from our participation with Oberon in the academics part of Project 7, an initiative launched by Microsoft Research with the goal of exploring language interoperability on the .NET platform. Our motivation to continue was double-faced: a) further explore the potential of .NET and b) experiment with evolutionary language concepts.

This report is structured three parts: a) a brief and illustrated conceptual introduction of the new programming language Zonnon, b) a discussion of the mapping of its object system to .NET and c) an overview of the development of a Zonnon compiler for .NET, with a special emphasis on the use of a new compiler integration technology called CCI.

2 The Zonnon Language

Zonnon is a new programming language. More precisely, it is an evolution of the Pascal, Modula, Oberon language line [1]. Its design has been guided by the following principal objectives:

- **Versatility in terms of programming paradigms**

The language should seamlessly cover a variety of programming paradigms and styles. If used as a tool for the development of software of any of the following categories, it should neither impose a methodological “corset” nor require crooked trains of thought:

- algorithms and data structures

- modular and embedded systems
- object-oriented systems
- component systems
- actor and agent systems
- **Uniform concept of abstraction**
The language should provide one generic notion of abstraction. It should refrain from conceptually distinguishing similar kinds of abstractions or abstractions whose differences are caused by artefacts of a lower system level.
- **Effective compositional framework**
The language should support and encourage the reuse of existing software components for the construction of new and more complex systems.
On the background of these design principles, we are now well prepared to present the concepts of the Zonnon language and the rationale of its constructs.
There are four different kinds of self-contained program entities in Zonnon: Objects, modules, definitions and implementations. The first two are runtime entities, the third is an entity of abstraction and the fourth is an entity of reuse. Here is a brief characterization:
 - **Object**
Template of a self-contained runtime entity. May include intrinsic behaviour expressed in terms of one or more activities running as separate threads. Can be instantiated dynamically under program control in arbitrary multiplicity.
 - **Module**
Object whose creation is controlled by the system. Typically singleton.
 - **Definition**
Abstract description of a *facet* of an object. Typically containing declarations of types, constants, variables and method signatures (in this article we use the term *method* synonymously with *procedure*)
 - **Implementation**
Reusable partial or total implementation of a definition.
In the following sections, we take an exemplary approach to explaining these entities and their use in more detail and breadth, where we consciously choose caricature-like simplified illustrations.

2.1 Objects and Modules

Compared to Oberon, the notion of object is conceptually upgraded in Zonnon by one or more possible *intrinsic activities* that may develop concurrently without the object being called at all [2].

In some extraterrestrial world, the following kind of *Zoilath* creatures may try to survive. While the temperature is below a certain minimum, the instances of this species simply hibernate, otherwise they either take a random walk or, if they are hungry, hunt for prey.

```
OBJECT Zoilath (x, y, t: INTEGER);
  VAR X, Y, temp, hunger, kill: INTEGER;
  PROCEDURE SetTemp (dt: INTEGER);
```

```

        BEGIN { EXCLUSIVE } temp := temp + dt
    END SetTemp;
    ACTIVITY;
    LOOP
        AWAIT temp >= minTemp;
        WHILE hunger > minHunger DO
            HuntStep(5, kill);
            hunger := hunger - kill;
            WHILE (kill > 0) & (hunger > 0) DO
                HuntStep(7, kill);
                hunger := hunger - kill
            END;
            RandStep(2)
        END;
        RandStep(4); hunger := hunger + 1
    END
END
BEGIN X := x; Y := y; temp := t; hunger := 0
END Zoilath;

```

Note that the *activity* part of the object declaration coherently tells the full life story of these creatures. Further note that their behavior still depends crucially on the weather makers in the environment calling the `SetTemp` method (in mutual exclusion, of course). In particular, every instance entering the loop is blocked by the `AWAIT` statement until the temperature is reported to have risen above the limit. Further note that the initialization statements in the object body guarantee each creature to start its active life in a well-defined state, depending on the object parameters passed by the creator.

While Zoilaths may exist in large number in our hypothetical terrarium, and new instances can be created dynamically at any time, there is only one environment. However, this does not mean that it is an unstructured monolithic block. Quite the reverse, the environment is typically a conglomerate of separate parts or *modules*, where each module roughly corresponds to a certain service. Clearly, some services may *import* (use) others. For example, assume that modules *Food*, *Map* and *Weather* exist as part of our environment, where *Food* imports *Map* and *Weather*, and *Map* imports *Weather*:

```

MODULE Food;
    IMPORT Map, Weather;
    PROCEDURE { PUBLIC } GetLocation (VAR X, Y: INTEGER);
    BEGIN
        IF Weather.isWet(X, Y) THEN Map.GetLake (X, Y)
        ELSE Map.GetMountain (X, Y)
        END
    END GetLocation;
END Food;

MODULE Map;
    IMPORT Weather;

```

```

VAR ...
PROCEDURE GetNextLake (VAR X, Y: INTEGER);
PROCEDURE { PUBLIC } GetLake (VAR X, Y: INTEGER);
BEGIN GetNextLake(X, Y);
  WHILE Weather.isIcy(X, Y) DO GetNextLake(X, Y) END
END GetLake;
PROCEDURE { PUBLIC } GetMountain (VAR X, Y: INTEGER);
END Map;

MODULE Weather;
  PROCEDURE { PUBLIC } isWet (X, Y: INTEGER): BOOLEAN;
  PROCEDURE { PUBLIC } isIcy (X, Y: INTEGER): BOOLEAN;
  ...
END Weather;

```

The `PUBLIC` modifier denotes visibility of the item to importing clients. If absent (as in the case of `GetNextLake` in module `Map`), the item is hidden to clients. Figure 1 gives a graph representation of the modular system just discussed.

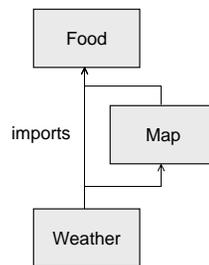


Fig.1. Example of a modular system

Modules in *Zonnon* are structural parts of the runtime environment, appearing in the clothes of ordinary (possibly active) objects. In combination with the *import* relation, the module construct is a powerful system structuring tool that we badly miss in most of the popular object-oriented programming languages. As a side remark we add that module graphs should neither be confused with object-oriented inheritance diagrams nor with runtime data structures.

2.2 Definitions and Implementations

A *definition* is an abstract view on an object from a certain perspective or, in other words, an abstract presentation of one of its *facets*. For example, a jukebox has two facets. We can look at it alternatively as a record store or as a player. The corresponding definitions are:

```

DEFINITION Store;
  PROCEDURE Clear;
  PROCEDURE Add (s: Song);
END Store.

```

```

DEFINITION Player;
  VAR current: Song;
  PROCEDURE Play (s: Song);
  PROCEDURE Stop;
END Player.

```

Assume now that we have the following *default implementation* for the Store definition

```

IMPLEMENTATION Store;
  VAR rep: Song;
  PROC Clear;
    BEGIN rep := NIL
  END Clear;
  PROC Add (s: Song);
    BEGIN s.next := rep; rep := s
  END Add;
BEGIN Clear
END Store.

```

We can now aggregate the Store implementation with the jukebox object:

```

OBJECT JukeBox IMPLEMENTS Player, Store;
  IMPORT Store; (* aggregate *)
  PROCEDURE Play (s: Song); IMPLEMENTS Player.Play;
  PROCEDURE Stop IMPLEMENTS Player.Stop; ...
  PROCEDURE { PUBLIC } AcceptCoin (value: INTEGER);
END JukeBox.

```

It is worth noting that an arbitrary number of implementations can be aggregated with an object. Further we emphasize that implementations need not be complete but may leave methods unimplemented. The rule is that, in this case, it is mandatory for the implementing object to provide an implementation, while in all other cases (re-)implementation is optional.

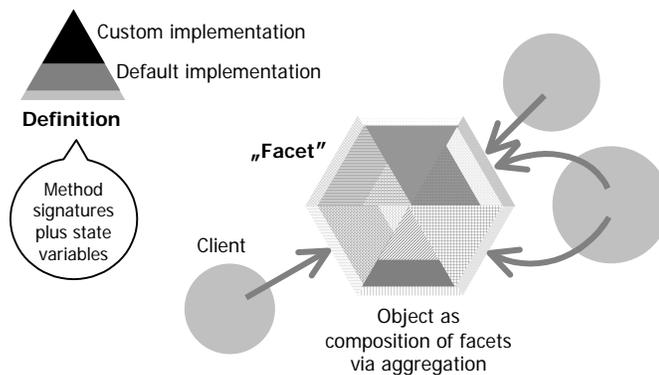


Fig.2 Definition: unifying abstraction concept

Figure 2 depicts our object model. Zonnon objects present themselves to external observers as multi-faceted entities, with a *home facet* and a set of *remote facets* that are specified in form of definitions. Internally, Zonnon objects consist of a kernel, possibly including active behavior specification, and an implementation for each definition. Every implementation splits into a default part and an object-specific part, where the default part is simply aggregated with the object.

Depending on their perspective, clients access Zonnon objects either directly via some public method (corresponding to the home facet perspective) or indirectly via definition. For example, if

```
VAR j: Jukebox; s: Song;
```

is a declaration of a Jukebox instance and a song instance, then `j.AcceptCoin(5)` is a direct access to the Jukebox home facet, while `Player(j).Stop` and `Store(j).Add(s)` are indirect accesses to `j` via definitions.

Declarations of object instance variables are either *specific* or *generic*. The above declaration of a juke box is specific. A corresponding generic variant would simply declare the juke box as an object with a postulated implementation of the definitions `Player` and `Store`:

```
VAR j: OBJECT { Player, Store } ;
```

Unlike in the above case of a specific declaration, it would obviously not make sense to address the Jukebox's home facet.

Let us conclude this section with some additional comments including fine points and future perspectives.

- As a fine point we first note that there need not be a one to one correspondence between definitions and implementations. It is well possible for an object to expose a definition without aggregating any default implementation. In turn, an object may aggregate an implementation without exposing the corresponding definition (if any exists at all).
- Our concept of definition and implementation subsumes and unifies two concepts of abstraction that are separated artificially in languages as Java and C#: Base class and interface. While the base class abstraction is generously supported by these languages in the form of implicit inheritance, the interface abstraction is not, and its use necessitates the delegation of the work to "helper classes". We consider this asymmetry a highly undesirable artifact from the perspective of software design, because it requires the designer to distinguish one abstraction as the "primary" one. What is the primary abstraction in the case of our juke box, what in the cases of objects that are both container and applet, figure and thread etc ?
- Our model of aggregation resembles multiple inheritance as used in some object-oriented languages such as C++ and Eiffel. However, there are two significant differences. First, our implementation aggregates are never instantiated autonomously in the system, they correspond to abstract classes. Second, our inheritance is explicit by name rather than implicit, protecting the construction from both name conflicts and obscure or unexpected method selection at runtime.
- The concept of definition is a principle rather than a formal construct. It is a door to extending the language without compromising its core part. Two examples:

- We may liberally introduce a *refinement* relation between definitions that, in its simplest form, reduces to extension. However, more elaborate forms might support stepwise refinement by allowing entire refinement chains of some component specification.
- In future versions of the language, we may open the syntax and accept, for example XML element specifications as definitions. On this basis it would definitely be worth revisiting XSLT transformation schemes and “code behind” techniques for active server pages.

2.2 Small Scale Programming Constructs

With the design of the Zonnon language, we have intentionally concentrated on the large scale aspects of programming and have largely kept untouched the basic syntax and style of the Modula/ Oberon family.

With one exception, however: we added a *block statement* with optional processing modifiers in curly braces that allows us to syntactically unify similar but semantically different cases. For example, method bodies and critical sections of activities within an object scope both use the EXCLUSIVE modifier:

```
BEGIN { EXCLUSIVE } ... END
```

Other, more future oriented modifiers are CONCURRENT and BARRIER to signal the option of statement-parallel processing within the containing statement block. And finally, the block construct allows us to include *exception catching*, a first and important concession to interoperability on the .NET platform, a topic that we shall continue to discuss in the next chapter:

```
BEGIN { ... } ... ON exception DO ... END
```

3 Mapping Zonnon to .NET

While we consciously kept the design of Zonnon independent of the possible target platforms of its implementation, the aim of implementing the new language on the .NET platform can hardly be hidden. We already mentioned *exception catching* as a concession to interoperability. Another one is *namespaces*. However, unlike C#, Zonnon does not know an extra namespace construct. It simply allows qualified names to occur everywhere in the global program scope, where all but the last part are used to identify the name space.

A much more intricate topic is the smooth integration of Zonnon’s object model into .NET or, equivalently, its mapping to the .NET *Common Type System (CTS)* [3]. The following sections are devoted to this concern. They give a short, again example-oriented idea of our approach. Focusing on the non-standard part of the object system, the constructs that essentially remain to be mapped are *definition*, *implementation*, *object activity*, and *module*.

3.1 Mapping Definitions and Implementations

Different mapping options exist. If state variables in definitions are mapped to *properties* or *virtual fields* (given the latter exist), the complete state space can theoretically be synthesized in the implementing object, albeit with some access efficiency penalty. In contrast, the solution suggested below in terms of C# (.NET's canonical language) is based on an internal helper class that provides the aggregate's state space.

```
DEFINITION D;  
  TYPE e = (a, b);  
  VAR x: T;  
  PROCEDURE f (t: T);  
  PROCEDURE g (): T;  
END D;  
  
IMPLEMENTATION D;  
  VAR y: T;  
  PROCEDURE f (t: T);  
  BEGIN x := t; y := t  
  END f;  
END D;
```

is mapped to

```
interface D_i {  
  T x { get; set; }  
  void f(T t); T g (); }  
  
internal class D_b {  
  private T x_b;  
  public enum e = (a, b);  
  public T x {  
    get { return x_b };  
    set { x_b = ... } } }  
  
public class D_c: D_b {  
  T y;  
  void f(T t) {  
    x_b = t; y = t; } }
```

3.2 Mapping Object Activities

Mapping object activities is more a technical than a conceptual problem. Obviously, .NET multithreading facilities contained in the *System.Threading* namespace must in this case serve as images of the Zonnon's active constructs. Still, the mapping of the *AWAIT* statement is rather intricate. The suggestion in the table below is based on a mass notification of waiting objects at the end of each critical section. We are confident that we will be able to refine this "brute force" solution later.

ACTIVITY; S END	Method void act() { S }; Field Thread thread;
BEGIN ... END	x.thread = new Thread(new ThreadStart(act)); x.thread.Start();
AWAIT c	while (!c) { Monitor.Wait(this); }
BEGIN { EXCL } S END	Monitor.Enter(this); S; Monitor.PulseAll(this); Monitor.Exit(this);

3.3 Mapping Modules

Essentially, modules are simply mapped to “static” classes, that is, to classes with static members only. Here is a caricature of a window manager module programmed in Zonnon and a sketch of its .NET image:

```

MODULE System.WindowManager;
  IMPORT System.DisplayManager;

  OBJECT { VALUE } Pos;
    VAR X, Y, W, H: INTEGER
  END Pos;

  DEFINITION Window;
    VAR pos: Pos;
    PROCEDURE Draw ();
  END Window;

  VAR { PRIVATE } bot: OBJECT { Window };
    W, H: INTEGER;

  PROCEDURE Open (me: OBJECT { Window }, p: Pos);
  BEGIN ...
  END Open;

  PROCEDURE Change(me: OBJECT { Window }, p: Pos);
  BEGIN ...
  END Change;

BEGIN (* module initialization *) bot:=NIL;
  W := System.DisplayManager.Width();
  H := System.DisplayManager.Height()
END WindowManager.

```

```

namespace System {
    namespace Map {
        public struct Pos { ... };
        public class Window {
            public Pos pos;
            public virtual Draw ();
        }
        public sealed class WindowManager {
            private static int W, H; Window bot;
            public static Open (Window this;Pos p)
                { ... };
            public static Change(Window this;Pos p)
                { ... }
            public static void WindowManager () {
                ...
                W :=
System.DisplayManager.DisplayManager.Width();
                ...
            }}
        }
    }
}

```

4 The Zonnon for .NET compiler

4.1 Compiler Overview

The Zonnon compiler has been developed for the .NET platform and itself runs on top of it. It accepts Zonnon compilation units and produces conventional .NET assemblies containing MSIL code and metadata.

There are two versions of the compiler: a command-line interface version and one integrated into the Visual Studio (VS) environment.

The compiler is implemented in C# using the Common Compiler Infrastructure framework (CCI), designed and developed by Microsoft.

4.2 The Common Compiler Infrastructure

The CCI framework is a set of programming resources (C# classes) providing implementation support for compilers and other language tools for the .NET platform. In particular, the CCI provides support for building up an intermediate data structure from the source code and for successively transforming this representation into MSIL code. It also supports full integration of the compiler and of other components such as editor, debugger, project manager, online help system etc into Visual Studio.

The CCI framework should be considered as a subset of the .NET framework, because the namespace `Compiler` containing the CCI resources is included to `System` namespace. Its three major parts are

- An intermediate representation support
- A set of transformers or visitors
- An integration service

Intermediate Representation Support is a rich hierarchy of C# classes (henceforth called IR), representing the most common and typical constructs of modern programming languages. IR is based on the C# language architecture: its classes reflect the entirety of C# and CLR including constructs like *class*, *method*, *statement*, *expression* etc. This allows compiler developers to represent similar concepts of their language directly. In case the language features constructs that are unrepresented by IR, it is possible to simply add new IR classes. However, this must be accompanied by adding corresponding transformers – either as an extension of a standard “visitor” (see below) or as a completely new visitor.

Transformers or Visitors is a set of base classes performing consecutive transformations from the intermediate representation to a .NET assembly. There are five standard visitors predefined in CCI:

- Looker
- Declarer
- Resolver
- Checker
- Normalizer

All visitors walk through an intermediate representation performing various kinds of transformations. The *Looker* visitor (together with its companion *Declarer*) replaces *Identifier* nodes with the members or locals they resolve to. The *Resolver* visitor resolves overloading and deduces expression result types. The *Checker* visitor checks for semantic errors and tries to repair them. Finally, the *Normalizer* visitor prepares the intermediate representation for serializing it to MSIL and metadata.

All visitors are implemented as classes inheriting from CCI’s *StandardVisitor* class. Alternatively, the functionality of a visitor can be extended by adding methods for the processing of specific language constructs or a new visitor can be created. The CCI requires that all visitors used in a compiler (directly or indirectly) inherit from *StandardVisitor* class.

Integration Service is a variety of classes and methods providing integration into the Visual Studio environment. The classes encapsulate specific data structures and functionality that are required for editing, debugging, background compilation etc.

4.3 The Zonnon Compiler Architecture

Conceptually, the organization of our compiler is quite traditional: the *Scanner* transforms the source text into sequence of lexical tokens that are accepted by the *Parser*. The Parser is implemented by recursive-descent. It performs syntax analysis and builds an abstract syntax tree (AST) for the compilation unit using IR classes. Every AST node is an instance of an IR class. The semantic part of the compiler

consists of a series of consecutive transformations of the AST built by the Parser to a .NET assembly. Figure 3 shows the Zonnon compilation model.

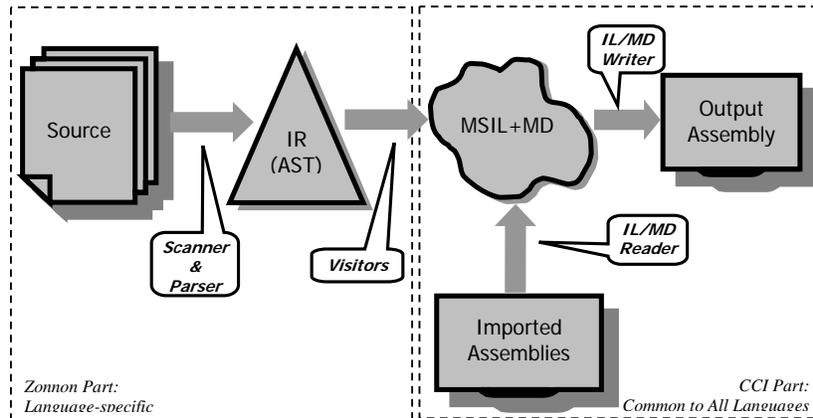


Fig. 3. Zonnon compilation model.

From an architectural point of view, the Zonnon compiler differs from most of the “conventional” compilers. The Zonnon compiler is no “black box” but presents itself as an open collection of resources. In particular, data structures such as “token sequence” and the “AST tree” are accessible (via a special interface) from outside of the compiler. The same is true for compiler components. For example, it is possible to invoke the Scanner to extract tokens from some specified part of the source code, and then have the Parser build a sub-tree for just this part of the source. Figure 4 illustrates the overall Zonnon compiler architecture.

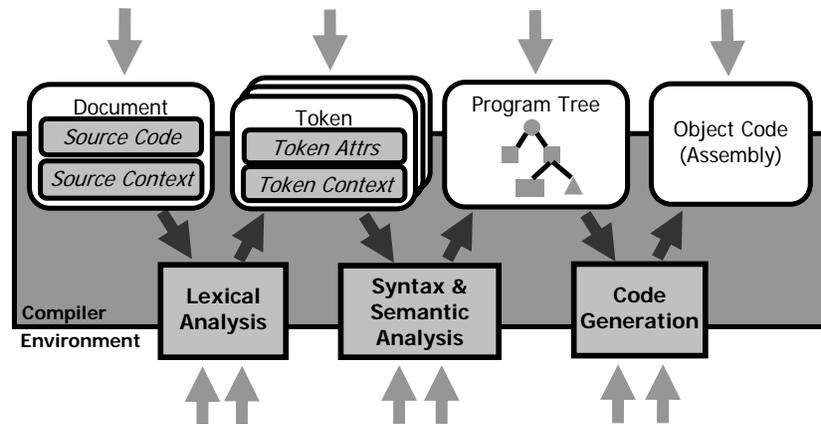


Fig. 4. Zonnon compiler architecture: compiler as a collection of resources.

This kind of open architecture is used by the CCI framework to provide a natural integration of the compiler into the Visual Studio environment. The CCI contains prototype classes for Scanner and Parser. The actual implementations of Scanner and Parser are in fact classes inheriting from the prototype classes.

Our compiler extends the IR node set by adding a number of Zonnon-specific node types for notions such as *module*, *definition*, *implementation*, *object* and for some other constructs having no direct counterpart in the CCI. The added nodes are processed by the extended *Looker* visitor which is a class inherited from the standard CCI *Looker* class. The result of the extended *Looker's* work is a semantically equivalent AST tree, however restricted to nodes of predefined CCI types. In particular, it is the extended Looker that implements the mappings described in Section 3.

5 ACKNOWLEDGMENTS

The authors gratefully acknowledge the opportunity of collaborating with Microsoft in this project. In particular, we thank Herman Venter for providing the CCI framework and letting us be his Guinea pigs. Our sincere thanks also go to Patrik Reali for his valuable advice concerning the mapping of Zonnon definitions to .NET. And last but definitely not least, we greatly appreciate Brian Kirk's good naming taste. *Zonnon* could in fact turn out to be more than just a code name.

6 REFERENCES

1. Wirth, N., The Programming Language Oberon, Software - Practice and Experience, 18:7, 671-690, Jul. 1988
2. Gutknecht, J., Do the Fish Really Need Remote Control?, A Proposal for Self-Active Objects in Oberon, JMLC'97, p. 207-220
3. Gutknecht, J., Active Oberon for .NET: An Exercise in Object Model Mapping, BABEL'01, Satellite to PLI'01, Florence, IT, 2001