Zonnon Language Experiment, or How to Implement a Non-Conventional Object Model for .NET

Jürg Gutknecht
Computer Systems Institute
ETH Zürich, Switzerland
gutknecht@inf.ethz.ch

Eugene Zueff
Computer Systems Institute
ETH Zürich, Switzerland
zueff@inf.ethz.ch

1. INTRODUCTION

This is a report on a work in progress. The project emerged from our participation with Oberon (a descendant of Pascal and Modula-2) in Project 7 that was launched by Microsoft Research in 1999 with the goal of implementing an exemplary set of non-standard programming languages for the .NET interoperability platform. Our motivation for continuing was double-faced; a) explore the potential of .NET in combination with the new compiler integration technology CCI as an experimental field for language design and b) implement Zonnon for .NET, an evolution of Oberon for .NET.

2. ZONNON FOR .NET

Zonnon for .NET is a general purpose imperative programming language. It features a rich and powerful but highly uniform object model that supports a variety of programming styles, including a conventional algorithms-and-data-structure style, modular and object-oriented programming styles, and actor-based computing models.

The highlights of the object model are

- a unique concept of abstraction called *definition*
- a corresponding notion of default *implementation*
- a combination of conventional object and thread called *active object*
- a *module* construct

Roughly, definitions subsume and unify the common abstractions of super class and interface and, in combination with a mechanism of static aggregation of implementations, replace the concepts of class hierarchy and single inheritance. Active objects come with an integrated thread of control that describes their runtime behavior. Modules are objects with a system-controlled life-cycle.

In addition, a *block statement* with optional processing modifiers in curly braces and an exception catching clause has been added to the language. Typical processing modifiers are ACTIVE (run as a separate thread), EXCLUSIVE (run under mutual exclusion within the corresponding object scope), and CONCURRENT (all statements may potentially run concurrently).

We shall now illustrate the new constructs of the language by a small series of simple but typical examples.

2.1 Definitions and Implementations

A jukebox has two "facets". We can look at it alternatively as a record store or a player. Correspondingly, we have the following definitions:

```
DEFINITION Store;
  PROC Clear;
  Add (s: Lib.Song);
END Store.

DEFINITION Player;
  VAR cur: Lib.Song;
  PROC Play (s: Lib.Song);
  PROC Stop;
END Player.
```

Assume that we in addition have a default implementation for the `Store` definition

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

Then we get to this declaration of a jukebox object

```
OBJECT JukeBox IMPLEMENTS Player, Store;
  IMPORT Store; (* aggregate *)
  PROCEDURE Play (s: Lib.Song);
IMPLEMENTS Player.Play;
  PROCEDURE Stop IMPLEMENTS Player.Stop;
..
END JukeBox.
```

Note that the `Store` default implementation is aggregated implicitly to the object state space.

2.2 Active Objects

In a simple terrarium, the following kind of creatures may try to survive. If the temperature is below a certain minimum, the instances of this species simply hibernate, otherwise they either walk around randomly or, if they are hungry, they hunt for prey.

```
OBJECT Creature;
  VAR X, Y, temp, hunger, kill: INTEGER;
  PROCEDURE NEW (x, y, t: INTEGER);
    BEGIN X := x; Y := y; temp := t;
          hunger := 0
  END;
  PROCEDURE SetTemp (dt: INTEGER);
    BEGIN { EXCL } temp := temp + dt
  END SetTemp;
BEGIN { ACTIVE }
  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 Creature;
```

Note that the body part of the object declaration coherently tells the full life story of such creatures. Also note that their behavior still depends crucially on the environment calling the `SetTemp` method. In particular, any instance may be blocked by the AWAIT statement until the temperature is reported to have passed the limit.

2.3 Modules

Modules are system-wide objects whose life-cycle is managed by the system automatically. In particular, a module is loaded dynamically when it is first called. Modules are "static" objects that may statically import other modules.

Resource managers are good examples of system-oriented modules. The following sketch shows a window manager with encapsulated data structure that represents the current window configuration in the display space of the system. Note that the window manager is contained in a name space called *System* and that it relies on a second module called `DisplayManager`.

```
MODULE System.WindowManager;
  IMPORT System.DisplayManager;
                    (* static import *)
  OBJECT { VALUE } Pos;
    VAR X, Y, W, H: INTEGER
  END Pos;

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

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

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

  PROCEDURE Change(this:OBJECT{Window},p:Pos);
  BEGIN ...
  END Change;
BEGIN (* module initialization *) bot:=NIL;
  W := System.DisplayManager.Width();
                            (* delegation *)
  H := System.DisplayManager.Height();
END WindowManager.
```

A runtime system can be viewed structurally uniformly as an acyclic hierarchy of modules. Typically, the bottom-most and top-most members are system-modules and application-modules respectively.

3. MAPPING ZONNON TO .NET

A precondition for any language to be implementable for .NET is the existence of a mapping of its constructs to the Common Language Runtime (CLR). Depending on the paradigm and model represented by the language, this may be quite a challenge. In our case of an imperative language, the mapping of the executing part to the CLR execution engine is straightforward. What essentially remains is a specification of the mapping of definitions, implementations, active objects, and modules.

3.1 Mapping Definitions and Implementations

Different mapping options exist. If state variables in definitions are mapped to *properties* or *virtual fields* (given they exist), the complete state space can theoretically be synthetisized in the implementing object, however, with some efficiency penalty. In contrast, the solution below mapping to C# (.NET's canonical language) is based on an internal helper class providing 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 Active Objects

Mapping active objects is a rather technical than conceptual problem. Obviously, .NET multithreading facilities must serve in this case as images of the Zonnon active constructs. In the following we suggest a "brute force" approach to the mapping of the AWAIT statement. It is based on mass notification of waiting objects at the end of each critical section. We hope to be able to refine this solution later.

```
BEGIN { ACTIVE } S END

       Method void body() { S };
       Field Thread thread;

NEW(x)

       x.thread = new Thread(
         new ThreadStart(body));
       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 classes with static members only. Here is a sketch of the image of the above mentioned window manager under the .NET mapping:

```
namespace System {
  namespace WindowManager {
    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. ZONNON FOR .NET COMPILER

4.1 Compiler Overview

Zonnon compiler is developed for .NET platform and runs on top of it. Compiler accepts Zonnon sources (compilation units) and produces conventional .NET assemblies containing MSIL code and metadata.

There are two versions of the compiler: command-line compiler and compiler integrated into Visual Studio environment.

Compiler is implemented in C# using Common Compiler Infrastructure framework (see below), designed and developed in Microsoft Research, Redmond.

4.2 Common Compiler Infrastructure

Common Compiler Infrastructure (CCI) is a set of programming resources (C# classes) providing support for implementing compilers and other language tools for .NET platform. This support is not comprehensive; some aspects of compiler functionality are unsupported, e.g., lexical and syntax analysis – the details are given on the poster. But there is a very important thing CCI does support: integration into MS Visual Studio environment. Potentially, it is possible to achieve the full integration of a compiler with all VS components such as editor, debugger, project manager, online help system etc.

CCI framework could be considered as a part of entire .NET framework; the namespace `Compiler` containing CCI resources is included to `System` namespace.

CCI consists of three major parts: intermediate representation, a set of transformers, and integration service.

Intermediate Representation (IR) is a rich hierarchy of C# classes representing most common and typical notions of modern programming languages. IR hierarchy is based on the C# language architecture: its classes reflect all C# and CLR notions as class, method, statement, expression etc. This allows compiler developer to represent similar concepts of his/her language directly. In case the language has notions or constructs which are not represented by IR set of classes, it is possible to extend the original IR hierarchy adding new IR classes. Then corresponding transformations should be added – either as an extension of standard "visitors" (see below) or as a completely new visitor.

Transformers ("Visitors") is a set of based classes performing consecutive transformations from IR to .NET assembly. There are five standard visitors predefined in CCI: Looker, Declarer, Resolver, Checker, and Normalizer. All visitors walk an IR performing various kinds of transformations. Looker visitor (together with its companion Declarer visitor) replaces Identifier nodes with the members/locals they resolve to. Resolver visitor resolves overloads and deduces expression result types. Checker visitor checks for semantic errors and tries to repair them. Finally, Normalizer visitor prepares IR for serializing it to MSIL and metadata.

All visitors are implemented as classes inherited from CCI's StandardVisitor class. It is possible to extend the functionality of a visitor adding methods processing specific language constructs or create a new visitor. CCI requires that all visitors used in a compiler are (directly or indirectly) inherited from StandardVisitor class.

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

4.3 Zonnon Compiler Architecture

Conceptually, compiler organization is quite traditional: Scanner performs decoupling the source text into lexacal tokens which are accepted by Parser. Parser performs syntax analysis and builts the abstract syntax tree (AST) for the source compilation unit using

CCI IR classes. Every AST node is an instance of an IR class. “Semantic” part of the compiler consists of a series of consecutive transformations of the AST built by the Parser. The result of such transformations is the .NET assembly which is the result of compiler’s work.

However, from the architectural point of view, Zonnon compiler differs form most “conventional” compilers. Instead of the “black box” approach, where all compiler algorithms and data structures are encapsulated into the compiler and are not visible from outside, Zonnon compiler is, in fact, an open collection of resources. In particular, such data structures as token sequence and AST tree are acceptable (via special interface) from outside compiler. The same is for compiler components: for example, it is possible to invoke Scanner to extract tokens from a specified part of the source, and to invoke Parser to build a sub-tree for this part of the source.

Such architecture is stated by CCI framework and provides deep and natural compiler integration to Visual Studio environment. In order to support integration CCI contains prototype classes for Scanner and Parser. The actual implementation of Scanner and Parser components of the Zonnon compiler are classes inherited from those prototype classes.

Compiler extends IR node set adding a number of Zonnon-specific node types for the notions of Module, Definition, Implementation, Object and for some other constructs which do not have explicit prototypes among CCI nodes. The added nodes are being processed by the extended Looker visitor which is a class inherited from the standard CCI’s Looker class. The result of the extended Looker’s work is the semantically equivalent AST tree containing only nodes of predefined CCI types. Therefore, the extended Looker implements mappings described above in Section 3.

5. ACKNOWLEDGMENTS

The authors gratefully acknowledge the initiative taken by Microsoft for developing .NET and for inviting us to participate in Project 7. In particular, we thank Jim Miller, Brad Merrill, and Erik Meijer from Microsoft Research for numerous opportunities to discuss technical matters in connection with the development of Oberon for .NET, and Herman Venter for providing the CCI framework and letting us be his Guinea pigs. Last but not least, our sincere thanks go to Patrick Reali for his valuable suggestions concerning the mapping of Zonnon definitions to .NET.