The Zonnon Project: A .NET Language and Compiler Experiment

Jürg Gutknecht
Swiss Fed Inst of Technology (ETH)
Zürich, Switzerland
gutknecht@int.ethz.ch

Vladimir Romanov
Moscow State University
Computer Science Department
Moscow, Russia
romsrcc@rom.srcc.msu.su

Eugene Zueff
Swiss Fed Inst of Technology (ETH)
Zürich, Switzerland
zueff@inf.ethz.ch

ABSTRACT

Zonnon is a new programming language that combines the style and the virtues of the Pascal family with a number of novel programming concepts and constructs. It covers a wide range of programming models from algorithms and data structures to interoperating active objects in a distributed system. In contrast to popular object-oriented languages, Zonnon propagates a symmetric compositional inheritance model. In this paper, we first give a brief overview of the language and then focus on the implementation of the compiler and builder on top of .NET, with a particular emphasis on the use of the MS Common Compiler Infrastructure (CCI). The Zonnon compiler is an interesting showcase for the .NET interoperability platform because it implements a non-trivial but still "natural" mapping from the language's intrinsic object model to the underlying CLR.

Keywords

Oberon, Zonnon, Compiler, Common Compiler Infrastructure (CCI), Integration.

1. INTRODUCTION: THE BRIEF HISTORY OF THE PROJECT

This is a technical paper presenting and describing the current state of the Zonnon project. Zonnon is an evolution of the Pascal, Modula, Oberon language line [Wir88]. Major language concepts and some considerations concerning the system architecture were presented in a number of papers during the last two years [Gut02, Gut03].

The project emerged from our participation in Projects 7 and 7+, a collaboration initiative 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 part was Oberon for .NET, an interoperable descendant of Pascal and Modula-2.

The motivation for continuing the research was twofold: a) to explore the potential of .NET and in particular of the new compiler integration technology CCI and b) to experiment with evolutionary language concepts. The notion of *active object* was taken from the Active Oberon language [Gut01]. In addition, two new concurrency mechanisms have been added: an accompanying communication mechanism based on syntax-oriented protocols, borrowed from the Active C# project [Gun04], and an experimental "asynchronous" statement execution construct.

The new language was called Zonnon. It uses a compositional inheritance model. Typically, an object implements a specified set of *definitions*, each one accompanied by a default *implementation* that is aggregated into the object's state space. The syntax of Zonnon is presented in the [Zon05] document.

2. CURRENT STATE OF THE PROJECT

The core language is defined and stable but there are still ongoing experiments in the area of concurrency. The current compiler is a well-tested beta version. A specifically developed comprehensive Zonnon test suite containing more than 1500 Zonnon test cases and covering all language features is used for systematic testing of the compiler.

There are three user environments for the Zonnon compiler: command-line, Zonnon Builder and Visual Studio .NET. We note that, to the best of our knowledge, the Zonnon compiler is the first compiler

.NET Technologies'2005 *conference proceedings*,
ISBN 80-86943-01-1

developed outside of Microsoft that is fully integrated into Visual Studio. It is currently used in an experimental programming course for junior students in Nizhny Novgorod University, Russia [Ger05].

3. BRIEF INTRODUCTION TO ZONNON

Being a member of Pascal family of languages and thanks to a high degree of compatibility, Zonnon is immediately familiar to Modula/Oberon programmers. Most Oberon programs from the domain of algorithms and data structures are successfully compiled by the Zonnon compiler after just a few minor modifications.

However, from the perspective of "programming-in-the-large", Zonnon is much more elaborate compared to its predecessors. There are four different kinds of program units in Zonnon: *objects*, *modules*, *definitions* and *implementations*. The first two are program units to be instantiated at runtime, the third is a compile time unit of abstraction and the fourth is a unit of composition. Here is a brief characterization:

Object is a self-contained run-time program entity. It can be instantiated dynamically under program control in arbitrary multiplicity. Compared to Oberon, the notion of object is conceptually upgraded in Zonnon by the option of adding one or more encapsulated activities that control the intrinsic behavior of the object.

Module can be considered as a singleton object whose creation is controlled by the system. In addition, a module may act as a container of logically connected abstract data types and structural units of the runtime environment. In combination with the *import* relation, the module construct is a powerful system structuring tool that is missing in most modern object-oriented languages.

Definition is an abstract view on an object from a certain perspective or, in other words, an abstract presentation of one or more of its *facets*.

Implementation typically provides a possibly partial default implementation of the corresponding definition. It is a unit of reuse and composition that is aggregated into the state space of an object or module either at compile time or at runtime.

Zonnon also provides a novel object-oriented concurrency model that follows the metaphor of autonomous active objects interoperating with each other. The model incorporates encapsulated threads of activity serving two purposes: expressing intrinsic behavior and carrying out formal dialogs. Active C# provides a proof of concept for this concurrency model.

4. ZONNON MAPPINGS TO CLR

As mentioned before, the Zonnon object model differs from the virtual object model proposed by the .NET CLR. However, most Zonnon concepts can be mapped rather easily to corresponding CLR notions, with the help of a few minor tricks. The general approach taken was trying to make direct use of CLR high-level constructs rather than to optimize the Zonnon code image. In the following, we will consider some important mapping examples.

Zonnon **definitions** are represented as public interfaces, and their state variables are mapped to virtual properties. For example, the following sample definition

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

is mapped to the class:

```
public interface Random {
   System.Int32 Next { get; }
   void Flush(); }
```

Implementations are mapped to sealed classes with the same visibility as corresponding definitions. For example, a possible implementation of the random generator will look like as follows:

```
implementation Random;
   var { private } z : integer;
   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.
```

The compiler will generate code for this implementation that corresponds to the C# class:

```
public sealed class Random_implem : Random
{
   private System.Int32 z;
   System.Int32 Random.Next { get { ...; } }
   void Random.Flush ( ) { z = 3.1459; }
   public Random_Implem() { Flush(); } }
```

If no implementation is specified for a definition then the compiler generates a default implementation with

trivial properties. The example below illustrates this for the Random definition:

```
(*automatically generated definition companion*)
public sealed class Random_default : Random {
   System.Int32 Next_default;
   System.Int32 Random.Next {
            get { return Next_default; } } }
```

Zonnon **object** types actually behave like CLR classes and therefore are mapped to sealed classes with the same scope of visibility as the object type. In case a body is specified in an object type, it is mapped into an instance constructor as shown here:

```
object { public } R;
   var x : real;
begin
   ... x := 777.999;
end R.
```

```
public sealed class R {
   private System.Double x;
   public R ( ) {
      ... x = 777.999; ... }
}
```

The relationship "object implements definition" is a fundamental constituent in the Zonnon object model. It represents an obligation for an object type to provide the functionality promised by the definition. However, notice that a corresponding implementation (if it exists) is automatically imported by the compiler, and the object type needs to merely implement the missing parts and, if desired, to customize the default implementation. For example:

```
object R1 implements Random;
  (*implicitly imports Random implementation*)
  (* Procedure Next is reused from
      default implementation *)
  (* Procedure Flush is customized *)
  procedure Flush implements
                          Random.Flush;
  begin z := 2.7189; end Flush;
end R.
```

The "object implements definition" relationship is represented as a usual interface implemented by the class. To support the automatic reuse of the default implementation, its "class" image is aggregated into the class image of the object itself. Thus, the above object type shown will be represented as follows:

```
class R1: Random {
   private Random_implem implem;
   public System.Int32 Random.Next()
                    { return implem.Next(); }
   public void Flush() { z = 2.7189; } }
```

Finally, Zonnon **modules** are mapped to sealed classes (either public or internal, depending on the module's modifier) with static members, public static constructor (for the method body) and private instance constructor (to prevent uncontrolled creation of module instances) with empty body.

```
module Test;
   import Random;
     (* both definition and implementation
        are imported *)
   var x : object { Random };
     (* x's actual type is any type implementing
        Random *)
   object R2 implements Random;
  (*alternative random number implementation*)
   end R2;
begin
   x := new R1; ...
   x := new R2; ...
end Test.
```

5. THE ZONNON COMPILER

Compiler overview

The Zonnon compiler is written in C#. It accepts Zonnon program units and produces conventional .NET assemblies containing MSIL code and metadata. The Common Compiler Infrastructure (CCI) provided by Microsoft is used as a code generation utility and integration platform.

Technically the compiler is a single dll file that is directly integrated into Visual Studio and the Zonnon Builder environment, respectively. A small executable wrapper is added to make the command-line version of the compiler.

The Common Compiler Infrastructure

Conceptually, CCI provides three kinds of support for developing compilers for .NET (see Fig 5.1): high-level infrastructure (in particular, structures for building attributed program trees and methods for performing semantic checks on trees), low-level support (generating IL code and metadata), and programming service for integration.

From the programming perspective, the CCI is a set of C# classes that provide comprehensive support for implementing compilers and other language tools for .NET. In reality, the support is not fully comprehensive as, for example, lexical and syntactical analyses are left to the user. However, the CCI supports well the integration into Visual Studio (VS). With the support of CCI a full integration of a compiler with all VS components such as editor, debugger, project manager, online help system etc. becomes feasible.

The CCI framework should be considered as a part of the .NET framework, with the namespace *Compiler* containing the CCI resources included in the *System* namespace. It consists of three major parts:

intermediate representation, a set of transformers, and an integration service.

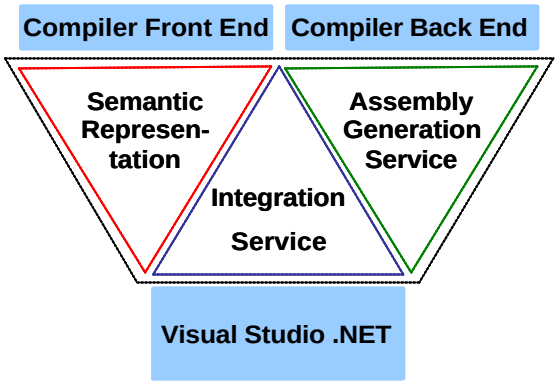

Figure 5.1 CCI Architecture

Intermediate Representation (IR) is a rich hierarchy of C# classes that represent typical constructs of modern programming languages. The IR hierarchy is based on the C# language architecture. Its classes reflect CLR constructs like class, method, statement, expression etc. plus a number of important notions not supported by CLR (e.g., nested and anonymous functions, or closures). This allows compiler developers to represent the corresponding concepts of their language directly in terms of a CLR class. In case a language feature is not presented by a CLR class, it is possible to extend the original IR class hierarchy. For each extension the corresponding transformations must be provided – either as an extension of a standard "visitor" (see below) or as a completely new visitor.

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

All visitors are implemented as classes inheriting from the CCI *StandardVisitor* class. It is possible to either extend the functionality of a visitor by adding methods for the processing of specific language constructs, or create a totally new visitor.

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

The Zonnon Compiler Architecture

Conceptually, the organization of the compiler is quite traditional: the *Scanner* transforms the source text into a sequence of lexical tokens that are accepted by the *Parser*. The Parser performs syntax analysis and builds an abstract syntax tree (AST) for the compilation unit using CCI 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. The result of such transformations is a .NET assembly.

It is worth noting that the Zonnon compiler does not make use of all CCI features. In particular, instead of extending the CCI Intermediate Representation by language-specific nodes, the compiler in fact creates its own Zonnon-oriented program tree in its first pass (see the data flow diagram in Fig. 5.2). The main reason for the extra tree is a clearer separation of the language-oriented and system-oriented compiler parts.

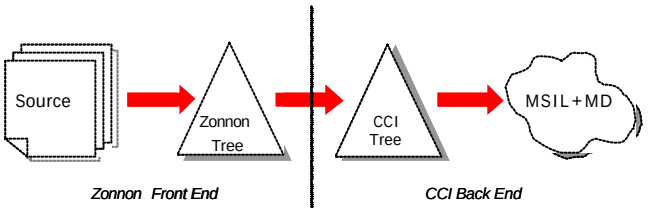

Figure 5.2 Compilation data flow

Also, the presence of two trees in the compiler reflects the conceptual gap between Zonnon and the CLR. It seems to be principally advantageous to represent information about Zonnon programs in a separate data structure that is independent of the target platform. Such a design leads to an optimal factoring of the compiler, with key tasks like name resolution and semantic control manipulating the Zonnon tree being totally independent of the CLR and .NET. Furthermore, the conversion from the Zonnon tree to the CCI tree explicitly implements and encapsulates the mapping from the Zonnon language model to the CLR Notice that functions logically related with both trees, the Zonnon tree and the CCI tree, are activated during the same compilation pass.

In the future the Zonnon tree will be extensively used for displaying structural information about Zonnon programs in VS' Solution Explorer views and for generating UML project diagrams by the Zonnon Builder (see Section 6).

From an architectural point of view, the Zonnon compiler differs from most "conventional" compilers.

In contrast to a "black box" approach whose goal is to hide algorithms and data structures, our Zonnon compiler presents itself as an open collection of resources. In particular, data structures such as "token sequence" and "AST tree" are exhibited to the outside world (via a special interface) for reuse by various programs. The same is true for algorithmic compiler components. For example, it is possible to invoke the Scanner to extract tokens from some specific part of the source code and then have the Parser build a sub-tree for just this part of the source.

Note that an analogous architecture is used by the CCI framework to support the deepest integration of any participating compiler with the Visual Studio environment. For example, the CCI contains Scanner and Parser prototype classes that served as base classes for the Zonnon parser and scanner components respectively.

6. THE ZONNON BUILDER

The Zonnon Builder is a conventional development environment comparable with many other IDEs. Our first goal in equipping the compiler with its own IDE was to provide an environment that looks familiar to Pascal programmers who are used to products like Delphi. On the other hand the Zonnon Builder can be considered as a simpler and light-weight alternative to full-featured environments like Visual Studio. The Zonnon Builder supports the full spectrum of a typical program development cycle, including source code editing, compiling, execution, testing and debugging. The Zonnon Builder supports structured projects consisting of several source files. Multi-file projects are compiled into a single assembly. It is possible to edit project files in different syntax-oriented editor windows simultaneously.

The second goal of the Zonnon Builder project was to offer a simple and comprehensible development environment for novices, specifically supporting the case of a simplified program development cycle in that a single program file is being developed, compiled, debugged and run. Such an option is very useful and convenient in an educational context.

The Zonnon Builder uses a special window to display compiler diagnostics. These are actually hyperlinks that can be clicked directly to visualize the part of the source code containing the (highlighted) error. In case of a program crash, the contents of the program stack are displayed in a separate window. The sections in the stack window are again hyperlinks (see Fig.6.1) and clicking at a section again causes the Builder to display and highlight the corresponding fragment of the source code.

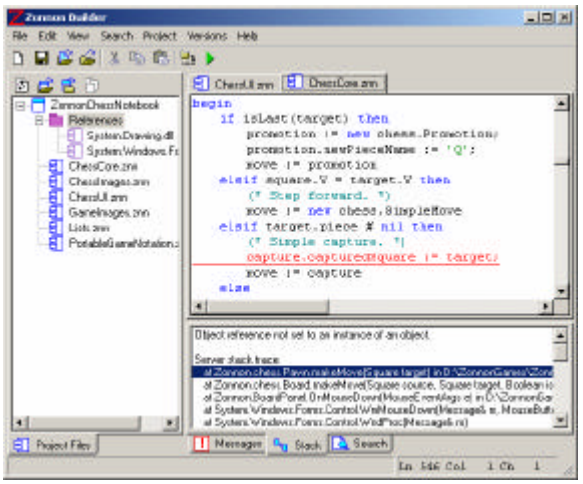

Figure 6.1 Zonnon program debugging

The Zonnon Builder also provides a simple version control mechanism. It is possible to save, restore and compare an arbitrary number of revisions for each project file (see Fig.6.2).

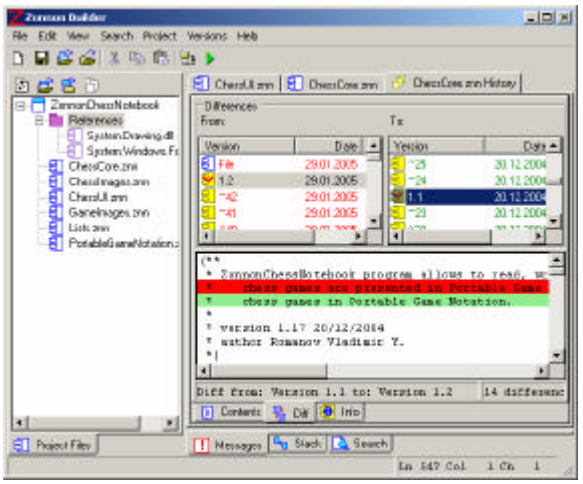

Figure 6.2 File versioning

Version control for the entire project is also supported. Each project version holds the state of all project files at a given time, together with an optional textual comment.

The Zonnon Builder Implementation

The Zonnon Builder as a whole is implemented in the form of a conventional .NET application. Its graphical user interface implementation reuses the standard .NET libraries *System.Drawing* and *System.Windows.Forms*. Some key components of the Builder such as the Zonnon-oriented program editor need to directly call the system API *(user32.dll)* because some functionality is missing in the .NET class libraries.

The design of the Zonnon Builder is intentionally kept largely independent of the specific programming language. Remaining dependences are encapsulated in two interfaces (see Fig.6.3).

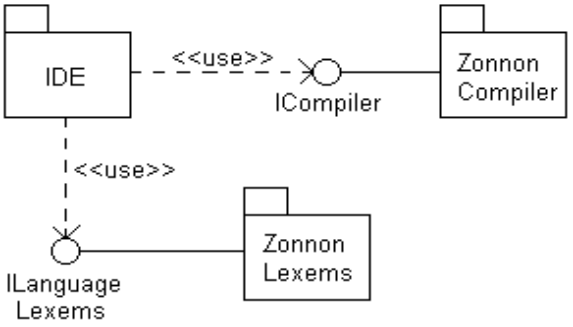

Figure 6.3 Zonnon Builder implementation

The *ICompiler* interface hides the implementation of the compiler. The Zonnon compiler wrapper implements the interface. The *ILanguageLexems* interface hides all language specific parts, for example the set of tokens. Therefore, it is easy to integrate any other programming language into the environment.

7. FUTURE WORK

Zonnon and Visual Studio

We aim at a closer integration with the Solution Explorer, including adequate interpretation of CLR notions (such as "type", "class", "method" etc.) in accordance with the semantics of the Zonnon language ("module", "definition", "procedure" etc.). We also strive for a closer integration of the object content presentation and the "intellisense" feature.

Zonnon Builder

The next Zonnon Builder version will include a code model for compiled Zonnon programs. Programs will be presented as a hierarchical tree whose nodes represent Zonnon compilation units and their contents, respectively. Another improvement will be automatic generation of UML diagrams for the static structure of Zonnon programs. The UML diagrams will visually present the different relationships between compilation units. Both presentation forms (code model and UML diagrams) will be integrated with the program text presentation. The integration with the standard CLR debugger is also planned.

8. LESSONS LEARNED

The experience in using the Zonnon language shows that it is quite convenient and can be used both for educational purposes (as the first programming language) and as an implementation tool. Some practical programs with non-trivial algorithms and graphical user interface were implemented in this language. The Chess Notebook program from the Zonnon web site is among the examples.

We are quite satisfied with the CCI framework. It is a well-designed, practical, powerful and flexible tool for building VS integrated compilers. It supports both the integration of existing compilers into the Visual Studio and the development of integrated compilers from scratch. CCI also can be considered as a more powerful and faster alternative to the *System.Reflection* library. The troubles with CCI were the lack of documentation and the unclear status of this framework.

9. CONCLUSIONS

Zonnon is the new programming language with a number of novel programming concepts and constructs. The language covers a wide range of programming models. This paper describes the current state of the Zonnon project: the language, the compiler and its development environment. The Zonnon compiler is also integrated into Microsoft's Visual Studio .NET environment.

The command-line Zonnon compiler, the Zonnon Builder, the Zonnon Language Report together with documentation and a large number of Zonnon program samples and tests are available on **www.zonnon.ethz.ch**.

10. ACKNOWLEDGMENTS

Our thanks go to Herman Venter, Brian Kirk, David Lightfoot, Alan Freed and to the first Zonnon users and programmers.

11. REFERENCES

[Ger05] Prof V.Gergel, personal communication.

[Gun04] R. Güntensperger and J. Gutknecht, Active C#, Proceedings of the 2nd International Workshop on .NET Technologies, Plzen 2004.

[Gut01] Gutknecht, J., Active Oberon for .NET: An Exercise in Object Model Mapping, BABEL'01, Satellite to PLI'01, Florence, IT, 2001.

[Gut02] J.Gutknect, E.Zueff, Zonnon Language Experiment, or How to Implement a Non-Conventional Object Model for .NET. OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.

[Gut03] J.Gutknecht, E.Zueff, Zonnon for .NET – A Language and Compiler Experiment. Joint Modular Languages Conference, JMLC2003, Klagenfurt, Austria, August 2003.

[Wir88] Wirth, N., The Programming Language Oberon. Software – Practice and Experience, 18:7, 671-690, Jul. 1988.

[Zon05] J.Gutknecht, E.Zueff, Zonnon Language Report, www.zonnon.ethz.ch.