Electronic Notes in Theoretical Computer Science 59 No. 1 (2001)
URL: `http://www.elsevier.nl/locate/entcs/volume59.html` 19 pages

Active Oberon for .NET: An Exercise in Object Model Mapping

Jürg Gutknecht [1,2]

Institute of Computer Systems
ETH Zürich
CH-8092 Zürich, Switzerland

Abstract

Active Oberon is a substantial evolution of the programming language Oberon. It distinguishes itself by a novel object model and by its integration into the .NET language interoperability framework.

The three concepts characterizing Active Oberon are: (a) *active object* types, (b) a single and unifying notion of abstraction called *definition*, and (c) a *static module* construct. These concepts are in fact powerful combinations of concepts: Active objects integrate active behavior with reactive message handling, definitions unify the units of usage, implementation and inheritance, and modules represent both package and static object. The rigid concept of class hierarchy has been sacrificed in Active Oberon to a more flexible concept of *aggregation* that is based on a generalized IMPLEMENTS relation. The relations IMPORTS and REFINES are used to specify static module dependencies and to derive new definitions from existing ones respectively.

This article is a report on a work in progress. We divide our presentation into three parts: (a) A short recall of the history of programming languages developed at the ETH, (b) an extensive conceptual overview of Active Oberon's object model called the *Active Object System (AOS)*, (c) a discussion of the mapping of the AOS into the *Common Type System (CTS)* exposed by .NET.

1 ETH Programming Language History

Oberon [1] is the latest member of the Pascal family of compact programming languages. From its ancestors *Pascal* and *Modula-2* it inherits a concise, highly expressive and self-explanatory syntax, strictly enforced data types, and a module concept. In addition, Oberon features sub-typing and polymorphism

[1] Thanks to Microsoft Research
[2] Email: `mailto:gutknecht@inf.ethz.ch`

based on record type extension. Active Oberon [2] is an evolution of Oberon combining the "spirit of Oberon" with the newest component technology. We aim at using Active Oberon in lectures on both small-scale and large-scale programming. Active Oberon for .NET is a first full implementation of the new language.

2 The Active Object System

2.1 An Extended Concept of Object Types

Active Oberon for .NET propagates the "new computing model" [3] that is based on a dynamic population of communicating *active objects* or *agents.* The corresponding language construct is an upgraded variant of an object type declaration with an optional body for the specification of *intrinsic behavior.* By convention, the body of an *object type* is run automatically as a separate thread as a side effect of the creation of an instance.

Ordinary (passive) objects are considered as active objects with an empty intrinsic behavior. They often take the role of *shared resources.*

Although the members of a typical active object scenario are largely autonomous, they are rarely truly independent of each other. As an immediate consequence, synchronization is required that necessarily manifests itself in temporary "passivation" of individual objects. On this low level of abstraction, the exhaustive enumeration of passivation conditions is surprisingly short:

- Unsatisfied precondition for continuation
- Competition for shared resource object
- Lack of processors

From the perspective of the object concerned, the first two situations necessarily arise synchronously and are governed by Active Oberon language constructs, while the third situation may arise asynchronously and cause preemption. The language constructs governing the synchronous cases are

- AWAIT(c)
- BEGIN { EXCLUSIVE } ... END

The AWAIT statement takes a scope-local Boolean condition c and blocks continuation while c is unasserted. The block statement in combination with the EXCLUSIVE modifier automatically guarantees mutual exclusion within the block.

It is worth emphasizing at this point that synchronization in Active Oberon is completely managed by the system. This is different from notification-oriented synchronization that fully relies on a fair behavior of the involved peer threads.

2.2 A Unified Concept of Abstractions

Active Oberon introduces an abstract concept called *definition*. This roughly corresponds to an *abstract class* in C++, Java and C# or, in other words, to an interface with optional state space and optional method implementations.

Note that, unlike the languages mentioned do, Active Oberon syntactically separates the notions corresponding to concrete and abstract classes because, in spite of their formal simularity, they are embodiments of radically different principles. Abstract classes (definitions) are descriptions of abstract concepts to be processed at compile time, while concrete classes (object types) are descriptions of objects to be instantiated at runtime.

Two relations apply to definitions

- IMPLEMENTS
- REFINES

If an object type *implements* a definition it commits itself to implement all previously unimplemented methods of this definition and to either accept or override existing implementations. Every definition implemented by an object type represents a *facet* or *role* of this type, and the entirety of implemented definitions constitutes the type's *API* (programming interface). New definitions can be derived from existing ones by *refinement*, that is roughly by extension of state space, functionality, or implementation.

While definitions in Active Oberon appear as coherent facets or units of use to clients, their specification need not be self-contained but can be derived from a base definition by *refinement*. The full semantics of refinement is intricate and may well include topics like weakening preconditions and strengthening postconditions in the future. For the moment it suffices to think of refinement as adding variables, methods, and implementations to an existing definition.

Perhaps the most obvious advantage of our object model over traditional object oriented systems is *symmetry*. Facets, roles, service units etc. can uniformly be modeled as definitions in Active Oberon, ready for implementation in any combination by any concrete object type. In a traditional object-oriented language like Java, two basically different ways exist for a concrete class to build on an abstraction A, namely taking A as

- Subclassing A and inheriting from it
- Implementing A, possibly by delegating standard work to some "helper class"

Of course, unless multiple inheritance is supported, the superclass option is applicable to at most one "main" abstraction per class.

Now, while in some cases the distinction of a main abstraction is quite natural, it may be artificial in others. To give an example: assume that some GUI framework supports two abstractions *Figure* and *Sensor* (mouse-sensitive area) and an object type *Control*. Then the question arises if a control is primarily a figure that in addition happens to be a mouse-sensitive area or primarily a sensitive area that happens to be represented as a figure. Both *Figure* and *Sensor* qualify equally well as main abstraction and can equally well serve as base class of class *Control*. If *Figure* is chosen as base class, then *Sensor* must be modelled as interface to be implemented by *Control*, possibly assisted by some helper class *SensorImplementation*. If *Sensor* is chosen as base class, then *Figure* must be modelled as interface possibly plus a helper class *FigureImplementation*. Both of these asymmetric design patterns are definitely inferior to the symmetric Active Oberon pattern consisting of two definitions *Figure* and *Sensor* and an object type *Control* implementing these definitions.

The most negative aspect of enforced asymmetry is an inherent ambiguity and lack of a "canonical form" for the presentation and use of abstractions. This can easily be confirmed when reading, for example, any tutorial on concurrent programming in Java. The controversial choice is between (a) extending class *Thread* and (b) implementing interface *Runnable*. Even the most sophisticated explanations of a conceptual difference between (a) and (b) are much less convincing than the sheer pragmatic reason of using variant (b) whenever the class to be made concurrent already extends some other base class such as *Applet*.

A second significant advantage of our object model is the "light-weight" way of object usage. Clients need not be aware of the full "type" of a server object but merely need to know the signatures of the services they actually use. Obviously, such a minimum-knowledge model is particularly beneficial in cases that involve remote service objects whose full specification is locally unavailable.

We should note at this point that Active Oberon embodies an object model that in many ways resembles that of the original COM. In particular, the two models coincide in their interface-oriented object view and in the non-use of polymorphic typing and subclassing. However, they differ in the form of implementation support. While COM carefully avoids any form of implementation inheritance and refers to "hand-knitted" methods as *delegation* and *aggregation* instead, Active Oberon inherently supports multiple inheritance of pre-defined standard functionality at *compile time* (in contrast to languages like C++ and Eiffel supporting multiple inheritance at runtime). As a matter of fact, we could roughly characterize Active Oberon as "COM plus built-in aggregation".

Let us now get more technical and assume a configuration as caricatured in the following Active Oberon code.

```
DEFINITION I; (* pure interface *)
  PROCEDURE { ABSTRACT }  f (..); (*abstract method *)
END I;
DEFINITION D;
  VAR x: X; (* public state variable *)
  VAR { PROTECTED } y: Y; (* protected state variable *)
```

```
    PROCEDURE { ABSTRACT } e (..); (* abstract method *)
    PROCEDURE { ABSTRACT } f (..);

    PROCEDURE g (..): ..; (* standard implementation *)
      VAR u, v: U;
    BEGIN.. RETURN ..
    END g;

    PROCEDURE { FINAL } h (..); (* final implementation *)
    BEGIN .. f(..); ..
    END h;
  END D;

  DEFINITION D1 REFINES D;
    VAR { PROTECTED } z: Z; (* new protected state variable *)

    PROCEDURE e (..); (* standard implementation *)
    BEGIN .. x ..
    END e;

    PROCEDURE g (..); (* new standard implementation *)
    BEGIN .. RETURN D.g(..) + ..
    END g;

    PROCEDURE { ABSTRACT } k (..);(* new abstract method *)
  END D1;

  TYPE A = OBJECT IMPLEMENTS I, D1;
    ..
    PROCEDURE f (..); IMPLEMENTS I.f, D1.e;
    BEGIN .. D1.x ..
    END f;

    PROCEDURE f1 (..); IMPLEMENTS D1.f;
    BEGIN .. D1.e(..); ..
    END f1;

    PROCEDURE k (..); IMPLEMENTS D1.k;
    BEGIN
    END k;
    ..
  END A;
```

We immediately observe that

- x is the only public state variable in D and D1. y and z are *protected variables* whose access is restricted to refining definitions and implementing object types.

- A meets its obligations by providing implementations for I.f, D1.f and D1.k.
- A customizes the implementation of method D1.e.
- Within D1 and A qualification is used to identify members of a different scope. For example, D1.x and D1.e in A refer to variable x and to the standard implementation of e in D1.
- In this example, A IMPLEMENTS D1 implicitly implies A IMPLEMENTS D.

Let us now assume that a second refinement D2 of definition D exists and that A is supposed to implement both D1 and D2. Then we are faced with a more complex case of intersecting chains of refinement:

```
DEFINITION D2 REFINES D;
  PROCEDURE k (..);(* new method *)
  BEGIN ..
  END k;
END D2;

TYPE A = OBJECT IMPLEMENTS I, D1, D2;
  ..
  PROCEDURE f (..); IMPLEMENTS I.f, D1.e;
  BEGIN ..
  END f;

  PROCEDURE f1 (..); IMPLEMENTS D1.f;
  BEGIN ..
  END f1;

  PROCEDURE f2 (..); IMPLEMENTS D2.f;
  BEGIN ..
  END f2;

  PROCEDURE p (..); IMPLEMENTS D2.g;
  BEGIN ..
  END p;

  PROCEDURE k (..); IMPLEMENTS D1.k, D2.k;
  BEGIN ..
  END k;
  ..
END A;
```

In this case, the definitions implemented by A (D1 and D2) have a common root (D), and some semantic clarification is needed. First, we declare D's state space shared by D1 and D2 that is, x refers to the same instance of variable whereever it occurs in either D1 or D2. Second, we point out that the implication *A IMPLEMENTS D1* $\Rightarrow$ *A IMPLEMENTS D* is no longer unconditionally true. By convention, this implication holds if and only if, for each method of D, the identification of its implementation is unambiguous. In our example, A does not implement D because the identification of f's implementation is ambiguous. We note in passing that this convention is compatible with the more general semantic model based on "dominants" [4], [5] and adapted by C++, for example.

We can best illustrate this rule by an example. Assume that our sensor abstraction of above features a method *Handle*. Further assume that *ButtonSensor* and *PenSensor* are refinements of *Sensor* providing individual implementations of *Handle*. Finally assume that *ButtonPenControl* implements both refinements simultaneously. Then, a necessary and sufficient condition for *ButtonPenControl* to also implement the abstract *Sensor* definition is a reimplementation of *Handle* (most probably based on *ButtonSensor.Handle* and *PenSensor.Handle*).

Let us now briefly change our perspective and turn to the client's site. In accordance with the "light-weight" view of objects by clients are *polymorphic* declarations that emphasize the desired facets of servers rather than their full "type". For example,

```
a: OBJECT { D1, D2 }
```

defines an object variable or parameter a with guaranteed implementations of the definitions D1 and D2. And this would then be a possible scenario of use:

```
.. D1(a).f(..); D2(a).k(..); .. := D1(a).x; ..
```

Alternatively, a could be declared completely generically and be used under the safeguard of an IMPLEMENTS clause:

```
a: OBJECT
IF a IMPLEMENTS D1, D2 THEN
  .. D1(a).f(..); D2(a).k(..); .. := D1(a).x; ..
END;
```

Note that type-specific object variable declarations are still possible in Active Oberon. However, they are essentially used only in combination with *abstract data types* and external operators. For example,

```
TYPE T = OBJECT ... END;
PROCEDURE "@" (x: T): T ;
PROCEDURE "#" (x, y: T): T;
```

defines an abstract data type T featuring two operations @ and #.

2.3 A Concept of Static Modules

In combination with a static *import* relation the *module construct* is a simple but extremely useful tool for structuring large software systems according to the principle of separation of concerns. If augmented by guaranteed version compatibility between importing and imported modules, the module/import combination is in addition an extremely useful framework for maintaining large software systems.

In Active Oberon for .NET modules have a dual semantics of *package* and *static object*. More precisely, an Active Oberon module is both

- A package representing simultaneously a namespace and a compilation unit
- A static object together with three conventions of instantiation:
  - · The object initializer is represented by the module body
  - · As a precondition of instantiation, all imported modules (that is their static objects) must be created and initialized in a compatible version

For example, the following module declaration defines

- A package M consisting of three object types M.A, M.X and M.Y
- A static object M whose state variables are a, s and t, whose methods are F and G and whose initializer is the BEGIN ... END block at the end of the module text. In addition, the import clause requires modules L and N to be initialized and ready for delegation before M is instantiated.

```
MODULE M;
  IMPORT L, N;
  TYPE
    A = OBJECT { PUBLIC } END A;
    X = OBJECT { PUBLIC }
      VAR { PUBLIC } a: A;
      VAR x: X;
      ...
    END X;
    Y = OBJECT ... END Y;
  VAR { PUBLIC } a: A;
  VAR s, t: T;
  PROCEDURE { PUBLIC } F (t: T): A;
  BEGIN ...
  END F;
  PROCEDURE G (a: A);
  BEGIN ...
  END G;
BEGIN (* module initialization *)
    ... N ... (* delegate work to module N *)
END M.
```

By default, the elements of a module scope are private to the module. However, the PUBLIC modifier may be used to declare any module element as “exported”, that is as visible from the outside. In our example, the object types A and X are public, while Y is private. Further, the members a and F of the static object M are public, while s, t and G are private.

By convention, the members of object type scopes are private to the enclosing module. However, public object types may declare public members again by simply adding a PUBLIC modifier. For example, in object type X, state variable a is public, while x is private.

It is worth noting at this point that static module objects distinguish themselves from "ordinary" Active Oberon objects in only one point: They are created and managed by the system rather than by application programs. As a consequence, Active Oberon modules can implement definitions exactly as object types can and, correspondingly, they can equally be used by clients via definitions.

In concluding these explanations we emphasize that the system is expected to guarantee version compatibility between L, M, and N at both compile time and run time.

3 The Mapping to the Common Type System

3.1 Recapitulation of the .NET Interoperability Framework

From an interoperability perspective, the main constituents of the .NET language framework are

- **A component packaging model**
  - Compile-time naming hierarchy
    A *namespace* is a qualified set of names
  - Run-time deployment hierarchy
    An *application* is a logically connected set of assemblies described by a configuration script (in XML).
    An *assembly* is a deployment unit consisting of a collection of modules described by a manifest.
    A *module* is either a portable executable (DLL or EXE file) or a container for additional resources like fonts, pictures etc.
- **A virtual object system called CTS (Common Type System)**
  Supporting the well-known object concepts of *class, subclass, interface, static class member, instance class member, abstract method, virtual method, sealed (final) method, virtual call, static call* and *interface call.*
- **A stack engine model**
  Operating on an abstract execution stack.

MSIL is a comprehensive intermediate language covering all three .NET models so that, in the end, the task of implementing a language for .NET simply amounts to compiling the language to MSIL.

In the case of Active Oberon, the compiler uses a simple recursive descent strategy [6], with the benefit of a built-in mapping to the stack engine model. No substantial problems have arisen from this choice, and MSIL has proved as a very suitable target.

Returning now to a more conceptual level we first observe that languages within the .NET framework typically take a dual role as *consumer* and *producer* of components. Correspondingly, the primary tasks language implementers for .NET are confronted with can be summarized as

- Mapping their language to the .NET model (preferably to the official .NET common language subset CLS)
- Mapping the .NET model (at least the CLS) to their language

In the following sections we shall discuss these tasks in the special case of Active Oberon. For this purpose, we first recall the main constructs supported by Active Oberon and their connecting relations:

- Constructs: Module, object type, definition
- Relations: IMPORTS, IMPLEMENTS, and REFINES

The following table summarizes:

Construct	**Relation**	**Construct**
Definition	REFINES	Definition
Object type	IMPLEMENTS	Definition
Module	IMPLEMENTS	Definition
Module	IMPORTS	Module

Modules and definitions are the only compilation units in Active Oberon. Object types are not compiled separately, their declaration needs to be wrapped in some module scope.

3.2 Mapping Modules

Let us first concentrate on the compilation of modules and in particular on the compilation of our exemplary module M. The result is a cascade, an assembly M containing a portable executable M, containing (a) a collection of sealed classes A, X and Y corresponding to the object types declared in M and (b) a sealed class M corresponding to the static object associated with M.

The sealed class M is sometimes called the *module class.* It is static in the sense that all its methods are static. Note in particular how the module's initialization code is compiled into the constructor of the module class. In detail, the following activities are performed exactly once at instantiation time of the module class:

```
FOR each module N imported by M DO
  Check version of N;
  IF compatible with version expected by M THEN
    Load and initialize static object N
  ELSE throw loading exception
  END
END;
Initialize static object M
```

When compiling an Active Oberon module, the compiler implicitly generates a .NET namespace for later use by any .NET language. Concretely, in the case of our exemplary module M, a namespace M is generated containing members M.A, M.X, M.Y and M.M (the static object associated with M). The elements of the module object can be accessed by qualification as usual. For example, method F of M is referred to in general as M.M.F. Within Active Oberon, the shortcut notation M.F is considered equivalent with M.M.F.

Let us now turn to *definitions*, the second kind of Active Oberon compilation units. Remembering that definitions are abstractions it is natural to compile each definition into an abstract .NET class, with the immediate advantage that refinement (at least a simple form of it) harmoniously maps to class extension in .NET.

However, this approach is infeasible in view of the IMPLEMENTS relation, at least in the case of multiple definitions implemented by one and the same object type. The mapping of the IMPLEMENTS relation to .NET is in fact a most intricate problem. It essentially amounts to finding a mapping from a restricted form of "multiple inheritance" to single inheritance plus interfaces.

3.3 Mapping Definitions

For the sake of concreteness, let now A be an object type in Active Oberon, and let D be any definition implemented by A. We already know that A is mapped to a sealed .NET class. For the mapping of the IMPLEMENTS relation, we can basically identify three different modeling options:

- Declare D as base class of A
- Aggregate D with A
- Integrate D with A

We already mentioned the first option at the end of the previous section. It is simple to implement but is restricted to one definition per object type. The second and third options are replicable arbitralily.

In detail, several possibilities exist for *aggregating* D with A. Our favorite one is subsumed by a transformation of D into a pair consisting of an interface I_D and a static class C_D, constructed according to the following rules:

(i) Transform each state variable of D into a property signature, that is a pair (get, set) of abstract methods.

(ii) Define I_D as the set of property signatures constructed in (i) plus the method signatures of D.

(iii) Transform each method of D into a static method by (a) adding a reference parameter of type A to its signature and (b) consistently replacing accesses to D's state variables in the implementation by accesses to the corresponding properties of the object passed as reference parameter.

(iv) Define C_D as the set of the static methods constructed in (iii).

(v) Extend A's state space by D's state variables.

(vi) Have A implement I_D, that is implement the property signatures constructed in 1 and D's methods, either by explicit implementation or by delegation to C_D.

It is probably best to demonstrate this procedure with a concrete example. Let T be any data type, and let D and A be declared as

```
DEFINITION D;
  VAR x: T;

  PROCEDURE f (y: T);
    VAR z: T;
  BEGIN x := y ; z := x
  END f;

  PROCEDURE { ABSTRACT } g(): T;
END D;

TYPE
  A = OBJECT IMPLEMENTS D;
    PROCEDURE g (y: T) IMPLEMENTS D.g;
      VAR z: T;
    BEGIN z := x; x := y
    END g;
  END A;
```

Then, the result of the transformation introduced above formulated in C#, .NET's *canonical language*, is this:

```
public interface ID {
  int x { get; set; }
  void f(int y); /* implemented by SD.f */
  void g(int y);
}

public class SD {
  public static void f(D me, int y)
  { int z; me.x = y; z = me.x; }
}

sealed class A: ID {
  int D_x;
  public void g(int y) { int z; z = x; x = y; }
  public void f(int y) { SD.f(this, y); } /* delegation */
  public int x { get { return D_x; } set { D_x = value; } }
}
```

It is worth emphasizing that within the Active Oberon language domain

- The exact identity of each method called is known when A is compiled
- D is never instantiated and does not have to be precompiled

Consequently, our third modeling option relies on lazy code generation and takes D as a source code template to be *integrated* with A at compile time. The obvious advantage of this method is direct access to D's state variables without virtual calls, the disadvantage is unavoidable multiplication of runtime code.

Our current solution of preference is "optimized aggregation" or, more precisely, aggregation plus

- Mapping of one of the definitions implemented by A to A's base class
- Trusting the JIT optimizer to draw benefit from A's seal

3.4 Interoperability

Language interoperability is .NET's highlight and major strength, and language implementers on .NET are well advised to apply special care on both the *consumer perspective* and the *producer perspective* of their language.

Taking a producer perspective in our case of Active Oberon, we first recall that the compilation units are *modules* and *definitions*. From a compilation view, they share the following characteristics:

- Each compilation unit defines its own namespace
- The result of each compilation is an assembly

From the discussion in the sections 3.2 and 3.3 we can easily understand the following summary in tabular form:

Result from the compilation of a	**Outer shell**	**Inner shell**	**Contents**
Module	Assembly	PE Module	• One sealed class for each object type declared in the module • One static sealed class representing the static module object
Definition	Assembly	PE Module	• One sealed static class representing the set of preimplemented methods • One interface representing the signatures of the state variables and methods

In the interest of interoperability it could obviously be reasonable to offer options for

- Unwrapping the static module object from its name space
- Compiling definitions into abstract classes

Let us now switch to a consumer perspective. In an interoperable environment, the Active Oberon compiler must obviously be prepared to accept components produced by foreign languages. The abstract vehicle provided by .NET for this purpose are *namespaces.* Active Oberon supports the use of foreign namespaces by a USES clause. For example, the statement

- USES A, B, P.M

opens a symbolic portal to the namespaces A, B, P.M and allows the use of their ingredients without name qualification (except in cases of ambiguity) within the current module scope. A complication arises due to the fact that there is not in general a one-to-one correspondence between namespaces and deployment units. Command line hints specifying the assemblies required for the next compilation are a temporary solution.

However, the crucial questions from a consumer's view are

- What are the reusable components?
- How can they be used?

The answers are simple in essence: The reusable components are .NET interfaces and .NET classes, and they can be used by Active Oberon either as abstraction or as factory for the creation of service objects. The following table diversifies:

.NET component	**Active Oberon entity if used as abstraction**	**Active Oberon entity if used for instantiation**
Interface	Definition	–
Abstract class	Definition	–
Concrete class	Definition	Object type

Note as an important technical fine point that the definitions derived from foreign (non-Oberon) classes are precompiled and cannot be mapped to an *(interface, static class)* pair as explained in section 3.3. Instead, they must be mapped to a base class. As a consequence, at most one definition derived from a foreign class can be implemented per object type.

The following code excerpt sketches the implementation of a generalized 15-puzzle in Active Oberon for .NET. Module *Puzzle* defines two object types: *PuzzleForm* and *Launcher. PuzzleForm* implements (and inherits from) definition *System.Winforms.Form.* Because this definition is derived from a foreign class it must be mapped to *Puzzle.PuzzleForm's* base class. *Launcher* is an active object with an intrinsic behavior. The next section explains how such behavior is mapped to .NET.

```
MODULE Puzzle;
  USES System, System.Drawing, System.WinForms; (* namespaces *)
TYPE
  PuzzleForm = OBJECT IMPLEMENTS System.WinForms.Form;
    VAR nofElems: INTEGER; ...

    PROCEDURE MoveElem(pbox: System.WinForms.PictureBox;
      dx, dy, nofSteps: INTEGER);
      VAR i, j: INTEGER; loc: System.Drawing.Point;
      BEGIN ...
      END MoveElem;

      PROCEDURE TimerEventHandler(state: OBJECT);
      BEGIN ...
      END TimerEventHandler;

      PROCEDURE Dispose() IMPLEMENTS System.WinForms.Form.Dispose;
      BEGIN ...
      END Dispose;

      PROCEDURE NEW(image: System.Drawing.Image; (* constructor *)
        title: System.String; subDivisions: INTEGER);
      BEGIN ...
      END NEW;

    END PuzzleForm;

    Launcher = OBJECT
      VAR target: System.WinForms.Form;
      PROCEDURE NEW (t: System.WinForms.Form); (* initializer *)
      BEGIN target := t
      END NEW;
    BEGIN { ACTIVE }
      System.WinForms.Application.Run(target)
    END Launcher;

  VAR puzzle: PuzzleForm; launcher: Launcher; .

  BEGIN
    WRITELN("Puzzle implemented in Active Oberon for .net"); .
  END Puzzle.
```

3.5 Mapping Active Behavior

Probably the main profit we can gain from the active object construct in Active Oberon is its conceptual influence on the mindset of system builders. In terms of functionality, it is basically equivalent with object constructs that support the implementation of a *run()* method.

In particular, when compiling an active object type, we can easily mimic the active object construct in the .NET framework by proceeding as follows

- Map the object type's body to a *run()* method
- Augment the object type's constructor by statements creating and starting a *Thread* object that runs the *run()* method

Two active object topics still remain to be discussed: Mutual exclusion and assertion-oriented synchronization [1]. Regarding mutual exclusion, the mapping is simple:

- Translate each mutually exclusive block statement *BEGIN* { *EXCLUSIVE* } ... *END* within an active object scope into a critical section on .NET that is into *CriticalSection.Enter .. CriticalSection.Exit*

The mapping of assertion-oriented synchronization is more complex. Consider a statement AWAIT(c) within any object scope, where c is a local Boolean condition. Then, this would be one possible solution:

- Translate AWAIT(c) into *while not c* { *Wait()* }*;*
- Automatically generate *NotifyAll()* calls at suitable points, for example at the exits of mutually exclusive block statements.

Obviously, this topic will need further investigation and optimization, in particular when efficiency matters.

3.6 Language Fitting

We should not conclude this section on the mapping of Active Oberon to .NET without mentioning a few minor adjustments of the original Oberon language that should be regarded as a tribute to making the revised language fit optimally in the .NET interoperability scheme.

- **Delegates**. We extended Oberon's concept of procedure variables to *method variables* that are able to represent arbitrary method values of arbitrary object instances and are no longer restricted to global procedure values. No language changes were necessary. The compiler now simply allocates a pair of pointers *(code pointer, object base)* for every method variable.

- **Value types**. Besides of the ordinary reference types, the .NET object system also supports "value" types. In Active Oberon, value types are called *record types*. Record types are declared in form of the well-known RECORD ... END construct. Their members are called *fields*. Methods in record types are not supported. Nor is boxing.

- **Overloading**. We extended Oberon to support method overloading. In the case of ambiguity, we require calls of an overloaded method to explicitly specify the desired signature as, for example, in

  ```
  u := MyModule.MyOverloadedMethod{(S, T): U)}(s, t)
  ```

  with *s*, *t* and *u* of types *S*, *T* and *U*, respectively.

- **Base method calls**. Because Oberon does not support subclassing, no special construct is provided for base method calls. However, calls of method implementations in definitions are possible from within the implementing object type simply by using their exact and fully qualified names.

- **Exception handling**. Active Oberon provides a rough form of exception handling. The corresponding throw/catch pair of constructs is
  - · A built-in THROWEXCEPTION procedure
  - · An extended form of the block statement:

```
BEGIN (* regular code *)
  EXCEPTIONALLY (* summary exception handling code *)
END
```

- **Namespaces**. Oberon does not provide an explicit namespace construct. However, module packaging can nevertheless be achieved simply by using a qualified module name. For example, a module M that is a member of a module package P could be called P.M.

- **Assemblies**. Except in the case of Active Oberon modules, there is no guaranteed one-to-one correspondence between namespaces and deployment units in .NET. In Active Oberon, command line arguments serve as locating hints to the compiler. If the *System* namespace is imported, then *mscorlib* is automatically located.

- **Procedure nesting**. Because access to intermediate variables is unsupported by .NET, the current Active Oberon compiler does not allow nested procedures.

4 Summary and Conclusion

This is a report on a joint project with Microsoft Research whose goal is the implementation of Oberon as a research language on the .NET interoperability platform. We took the opportunity and slightly revised the original Oberon language and its object model. Our two conceptual main achievements are: (a) A unified notion of *active object* that subsumes (passive) objects and concurrency and (b) a unified concept of abstraction called *definition* that subsumes the concepts of base class and interface.

The current version of the Active Oberon for .NET compiler uses a simple and fast one-pass, recursive descent strategy. It is reflectively programmed in Active Oberon for .NET and is able to self-compile. It makes use of .NET's *System.Reflection* API for internalizing imported classes and interfaces but still produces textual MSIL code. Also, in some cases the current compiler relies on explicit hints provided by the programmer, "forward"-declarations being the prominent example.

We are currently developing a second version of the Active Oberon compiler. It is again based on the recursive descent parsing strategy but will generate a more elaborate intermediate data structure that can serve as a basis for the following improvements:

- Resolve mutual references in the source code without a need of forward declaration

- Generate native MSIL code via the *System.Reflection Emit* API
- Apply the definition mapping rules in their full generality

Also, in the course of a different, local project we are currently implementing Active Oberon on a custom flyweight system kernel called Aos [7], [8], running natively on Intel SMP and StrongARM systems. In the end this will allow us to compare the efficiency of the two runtimes and finetune our implementations.

Some preliminary benchmarks have shown the JIT optimizer to generate about 50% more efficient native Intel code in comparison with our native (non-optimizing) Oberon compiler.

In concluding, we emphasize that we have concentrated in this essay on the aspect of the object model because this topic is most intimately related with "programming-in-the-large" and with interoperability and because it still bears ample research potential. However, we should not forget that, quantitatively, the translation of executable statements into the stack engine is still the dominant part of the compiler. In this respect we have not much to criticize, the stack engine is well designed and allows our one-pass recursive descent algorithm to emit MSIL code naturally. A minor fine point is a missing instruction for swapping the two topmost elements on the stack. Such an instruction is beneficial for constant expression folding by the compiler.

A more radical and definitely worthwhile thought revolves around a new API for compiler writers based on a parse tree data structure instead of on MSIL code. Not only would such an API factor out the reflection parts from each compiler but it would also take the burden from compilers to maintain their own symbol table. As a net effect, language implementation on .NET would be quite easy, and doors could be opened for the implementation of hundreds of small special purpose languages.

5 Acknowledgement

My first thanks go to Microsoft Research and in particular to James Plamondon, Pierre-Yves Saintoyant, Brad Merrill, Jim Miller, Dan Fay, Erik Meijer, and Don Syme for launching, managing, and supporting this project, for inviting us to participate, and for hosting us in the devlabs.

My sincere thanks go also to my collaborators Thomas Frey, Philipp Kramer, Hanspeter Högger, Karel Skoupy and Ben Smith-Mannschott for their competent help with the implementation of the compiler and to Patrick Reali for pointing out an optimization of my previous solution for aggregating definitions.

Further, I highly appreciated the profitable comments by Andrew Kennedy and the reviewers of a previous version of this article. Last but not least I gratefully acknowledge Brian Kirk's vital interest in Active Oberon and his constructive comments.

References

[1] Wirth, N., *The Programming Language Oberon*, Software-Practice and Experience, 18:7, 671-690, July 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] Milner, R., *Computing and Communication – what's the Difference?*, The Computer Laboratory, Cambridge University, June 1999, Speech given at the occasion of Tony Hoare's retirement in Oxford.

[4] Stroustroup, B., Private Communication, ETH Zürich, November 2000.

[5] Ramalingam, G., and Srinivasan, H., *A Member Lookup Algorithm for C++*, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY, 10598, USA

[6] Wirth, N., *Theory and Techniques of Compiler Construction*, Addison-Wesley, Reading, April 1996.

[7] Muller, P., *A Multiprocessor Kernel for Active Object-Based Systems*, JMLC 2000, p. 263-277.

[8] Reali, P., *Structuring a Compiler with Active Objects*, JMLC 2000, p. 250-262.