Active Oberon for .net

White Paper

Jürg Gutknecht, June 5, 2001. gutknecht@inf.ethz.ch
http://www.oberon.ethz.ch/oberon.net

1. Abstract

*Active Oberon for **.net*** is an evolution of the programming language *Oberon*. It was designed in the context of a joint project with Microsoft Research whose goal is the integration of Oberon into the new **.net** interoperability platform. Highlights of Active Oberon for **.net** are (a) an extended notion of *object types*, including "active" objects running their own thread of control, (b) a powerful abstraction called *definitions* and representing *facets*, *units of use* and *units of inheritance*, (c) a *module* construct that simultaneously acts as name-space and singleton object, and (d) a versatile block construct for grouping statements that share common characteristics.

2. Introduction and Overview

Active Oberon for **.net** introduces an abstract concept called *definition*. A definition is an interface with an optional state space and with optional predefined method implementations. Two relations apply to definitions: *IMPLEMENTS* and *REFINES*. Object types may *implement* a definition by providing implementations of previously unimplemented methods and/or re-implementations of pre-implemented methods. Each definition implemented by a given object type represents a *facet* of the objects of this type, and the entirety of implemented definitions constitutes the type's *client interface*. New definitions can be derived from existing ones by *refinement* that is by extension of state space, functionality, or implementation. *OBJECT* is a *generic type* that can be used for polymorphic object declarations in variable sections and in parameter lists.

Active Oberon for **.net** propagates the "new computing model" (quoting Robin Milner, Oxford, 1999) that is based on a population of collaborating *active objects* or *agents*. To this aim, object declarations in Active Oberon for **.net** are optionally equipped with a specification of an intrinsic behavior that is run automatically as a separate thread after object creation.

In combination, the concepts of definition, implementation, refinement, generic object type and integrated thread fully define Active Oberon for **.net**'s object model. Note in particular that this model (a) does not make use of the notion of class hierarchy and (b) is compatible with the typical architecture of a *distributed system* and scales up to *remote objects* perfectly.

Another interesting structural highlight in Active Oberon for **.net** is the *module*. Potentially every module embodies both a *name-space* and a *singleton object* that is created automatically by the system on demand. More precisely, the name of a module simultaneously denotes the entirety of definitions and object types declared in its scope and the singleton object made from the module's global variables (state space) and its global procedures (methods).

Finally, Active Oberon for **.net** features (a) a versatile *block statement* with *modifier clause* and with an optional *ON EXCEPTION* part and (b) *enumeration types* along the Pascal line. The following sections of this report informally present the history, rationale and novel aspects of the Active Oberon for **.net** language in greater detail.

3. Language History

Oberon is a member of the Pascal language family. From its ancestors *Pascal* and *Modula-2* it inherits a compact, highly expressive, and self-explanatory syntax, strictly enforced data types, and a concept of modules and public view. In addition, the original Oberon language features polymorphism based on record type extension. *Active Oberon for **.net*** is an evolution of Oberon that is targeted to the Microsoft **.net** platform.

4. Project Goals

Our idealistic goal in the Active Oberon for **.net** project was the design of program language that maps naturally to the **.net** framework and that is simpler, more economical and more powerful if compared to existing object-oriented languages.

Our specific goal was a language that

- preserves the spirit of Oberon
- provides a rich object model based on the metaphor of collaborating agents
- disentangles different concerns like reuse, polymorphism, sub-classing etc.
- emphasizes the view of software development as implementation of predefined abstractions
- is able to interoperate with other languages on **.net** both as a consumer and as a producer

5. Language Concepts

5.1. Object Types

In the original Oberon language *record types* are used to express object classes. In Active Oberon for **.net** record types are relegated to statically allocated composites (without pointer reference) describing logically connected collections of fields. For dynamically created objects Active Oberon for **.net** provides explicit *object types*. They are declared within a type section of a module by their name and structure.

Three kinds of objects corresponding to three stages of evolution can be distinguished in Active Oberon for .net: *Records* (fields only), *passive* objects (fields and methods), and *active* objects (fields, methods, and activity), where the activity is syntactically expressed as type body.

This is an example:

```
TYPE
  Link = OBJECT
    VAR next: Link; x, y: REAL;
  END Link;

  Figure = OBJECT
    VAR next: Figure; first: Link;

    PROCEDURE NEW (path: Link); (* constructor *)
    BEGIN first := path
    END NEW;

    PROCEDURE Display (X, Y: REAL);
      VAR cur: Link;
    BEGIN
      IF first # NIL THEN cur := first;
        WHILE cur.next # NIL DO
          DrawLine(X + cur.x, Y + cur.y, X + cur.next.x,
            Y + cur.next.y);
          cur := cur.next
        END
      END
    END Display;

  END Figure;

  Movie = OBJECT
    VAR first, cur: Figure; X, Y: REAL; stopped: BOOLEAN;

    PROCEDURE NEW (orgx, orgy: REAL; video: Figure);
      (* creator & initializer *)
    BEGIN X := orgx; Y := orgy; first := video
    END NEW;

    PROCEDURE Stop;
    BEGIN stopped := TRUE
    END Stop;

  BEGIN {active}
    stopped := FALSE; cur := first;
    WHILE ~stopped DO
      cur.Display(X, Y); cur := cur.next (*cyclic list*)
    END
  END Movie;
```

These declarations define object types called *Link*, *Figure*, and *Movie*. *Link* is a pure record type with fields *next*, *x*, and *y*. *Figure* is a passive object type with a creator/initializer *NEW* and a method *Display*. *Movie* is an active object type representing simultaneously an object and a thread. Every newly created instance of *Movie* runs its local process immediately after initialization as a side effect of its creation.

Figure 1 depicts the computing model based on *active objects* or “agents”. The figure shows a sample population of self-managing active objects evolving over time. Note in particular that with this computing model no driver processes are needed to remote control the evolution of objects. Participating objects may communicate synchronously or asynchronously via method call.

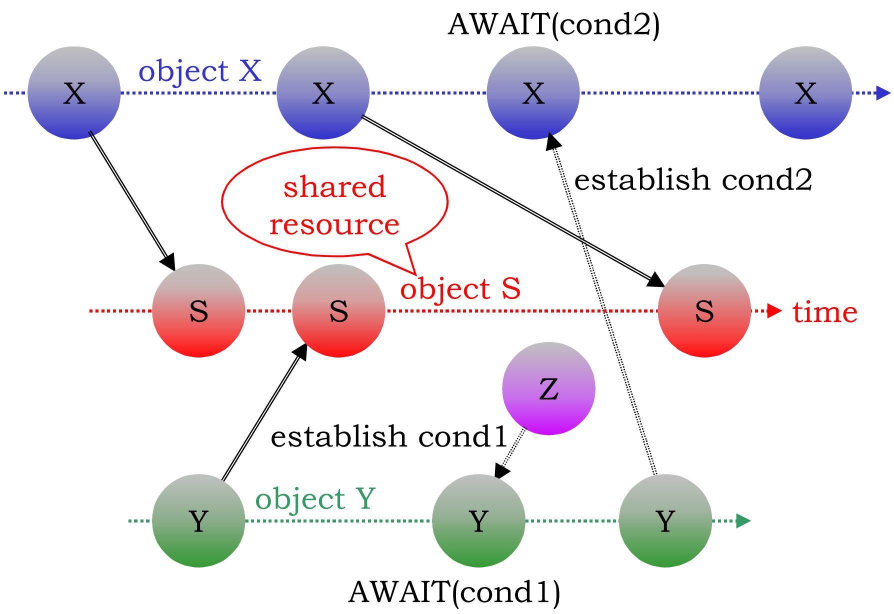

Figure 1. Sample Population of Communicating Active Objects

Remarks:

(1.) In Active Oberon for **.net**, object types are implicitly pointer based that is object instances are referenced and represented by a pointer. Active Oberon for **.net** does not provide explicit pointers or addresses.

(2.) Active Oberon for **.net** adapts the concept and notation of arrays from Oberon, with the addition of *dynamic arrays*. Dynamic arrays are declared with a star-symbol replacing the size specification. For example,

```
VAR a: ARRAY *, * OF T;
```

denotes a two-dimensional dynamic array that must be instantiated explicitly by *NEW(a, m, n);* where *m* and *n* are expressions representing the number of elements of type *T* in each dimension.

5.2. Definitions and Implementations

Definitions in Active Oberon for **.net** are abstractions representing an *interface*, together with optional state variables and pre-implemented methods. More technically, a general definition is a collection of field signatures, method signatures, and method implementations.

In order to meet special needs, definitions can be customized in Active Oberon for **.net** by *refinement*. Basically, refining a definition amounts to extending the set of state variables, methods, and method-implementations.

Before giving a concrete example, we need to explain two syntactic annotations distinguishing the role of a procedure declaration in a definition: { ABSTRACT } and { FINAL } specify procedure signatures without a following implementation (*abstract* methods) and procedures that are guaranteed not to be re-implemented in any refining definition and in any implementing object type (*final* methods) respectively. Procedure declarations without annotation specify methods with a *default* implementation.

We now come to a comprehensive example of a hierarchy of definitions of different kinds and an implementing object type: a pure interface *I*, a definition *D0* with a state space and a pre-implemented method, a refinement *D* of *D0* and an object type *T* implementing *I* and *D*. *D* newly defines *z* and *k*, implements *e*, re-implements *g*, and reuses *x*, *y*, *f* and *h*.

```
DEFINITION I; (* pure interface *)
  PROCEDURE { ABSTRACT } f (..); (*abstract method *)
  PROCEDURE { ABSTRACT } g (..);
  PROCEDURE { ABSTRACT } p (..): ..;
END I;

DEFINITION D0;
  VAR x: X; y: Y; (* state variables *)

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

  PROCEDURE g (..): ..; (* method with pre-implementation *)
    VAR u, v: U:
  BEGIN .. RETURN ..
  END f;

  PROCEDURE { FINAL } h (..); (* method with final implementation *)
  BEGIN .. f(..); ..
  END h;
END D0;

DEFINITION D REFINES D0;
  VAR z: Z; (* new state space variable *)

  PROCEDURE e (..); (* pre-implementation *)
  BEGIN ..
  END e;

  PROCEDURE g (..); (* new implementation *)
  BEGIN .. RETURN ..
```

```
  END g;
  PROCEDURE { ABSTRACT } k (..);(* new abstract method *)
END D;
```

Definitions serve both as units of usage and as units of inheritance. Each definition should be viewed as an individual *facet* of an object type exposed to its clients. If an object type *implements* a definition, it commits itself to implement all unimplemented (abstract) methods. In other words, clients of instances of this type can then count on a full implementation of the implemented definition.

In the following declaration, object type *T* commits to implement definitions *I* and *D*, and *t* is an instance of *T*:

```
TYPE T = OBJECT IMPLEMENTS I, D;
    …
    PROCEDURE f (..) ; IMPLEMENTS I.f, D.f ;
    BEGIN ..
    END f ;

    PROCEDURE a (..): .. ; IMPLEMENTS I.g ;
    BEGIN .. RETURN ..
    END a ;

    PROCEDURE myp (..) ; IMPLEMENTS I.p ;
    BEGIN
    END myp ;

    PROCEDURE b (..) ; IMPLEMENTS D.g ;
    BEGIN ..
    END b ;

    PROCEDURE k (..) : .. ; IMPLEMENTS D.k ;
    BEGIN
    END k ;
    …
END T;

VAR t: T;
```

A client who wants to make use of *t*'s interpretation of the services specified by *D* would then simply call *D*'s methods and fields safeguarded by *t*:

```
D(t).f(..); ..; .. := D(t).x;
```

In combination with refinement, clarification on the semantics of calls like *D(t).f(..)* is necessary. To this aim, let us now consider any refinement chain of definitions. Then the sequence of (re-)declarations of any given method along this chain must obey the following order, where "[..]" and "{ .. }" stand for option and repetition (including 0 times) respectively:

[abstract] { default } [final]

Assuming now that *D* is any definition, *f* a method of *D*, *T* an object type that implements *D*, and *t* an instance of *T*, we shall now give a precise operational semantics for method calls *D(t).f(..)*. First, tracking *D* back to its root base *D0,* we get a chain $D0 \subset .. \subset D \rightarrow T$ of refinements terminated by a (final) implementation. We then observe that the sequence of (re-)declarations of *f* along this chain must necessarily terminate with either a *default declaration* or a *final declaration*. In either case, we define the semantic meaning of *D(t).f(..)* by binding this terminal declaration to *D(t).f* (at compile time!).

A few remarks are in order at this point:

- If object type *T* implements a set of definitions with mutually disjoint refinement chains, then *T*'s state space is automatically augmented by every implemented definition's state space. Furthermore, *T* is implicitly said to implement every definition in every refinement chain.
- If object type *T* implements a set of definitions with intersecting refinement chains, then the state space of definitions belonging to the intersection is shared among all of the intersecting refinement chains. For example, if *T* implements *D1* and *D2*, and *D1* and *D2* both refine *D*, then *T*'s state space is extended by *D*'s state space and by *D1*'s and *D2*'s state space extensions respectively. In this case, *T* is said to implement *D* if and only if the binding of its methods is unambiguous, i. e. if the binding of *D*'s methods is the same along the chains *T -> D1 -> D* and *T -> D2 -> D*.

  For example, let us consider a *TaxPayer* definition with abstract method *PayTax*, a *USTaxPayer* refinement with concrete method *USTaxPayer.PayTax*, and a *FrenchTaxPayer* refinement, again with concrete method *FrenchTaxPayer.PayTax*. Then, if object type *WorldCitizen* implements *USTaxPayer* and *FrenchTaxPayer*, a necessary and sufficient condition for *DoubleCitizen* to also implement the abstract *TaxPayer* definition is a reimplementation of *PayTax* (most probably based on *USTaxPayer.PayTax* and *FrenchTaxPayer.PayTax*).
- Fields and methods of implemented definitions are explicitly qualified within the scope of the implementing object type. Therefore, name conflicts cannot occur.
- The notion of refinement of definitions has not been exploited completely up to now. For example, strengthening postconditions, weakening preconditions, narrowing type parameters (referring to the topic of parametric polymorphism) etc. would perfectly qualify for inclusion in the generic concept of refinement. In this connection it is worth mentioning that refinement is resolved completely at compile time in Active Oberon for **.net** because none of the participating definition can be instantiated.
- Our object model is compatible with interoperability within **.net**. All foreign classes from the **.net** language framework serving as units of inheritance for Active Oberon for **.net** are mapped to (precompiled) Active Oberon for **.net** definitions. Conversely, Oberon definitions and qualifying object types are mapped to (abstract) **.net** classes for further reuse, where an object type *qualifies* for reuse if and only if it implements at most one impure interface. Of

course, classes produced by Active Oberon for .net can unconditionally be used for delegation.

Figure 2 is a comprehensive illustration of Active Oberon for **.net**'s object model. It shows the following ingredients within the Active Oberon for **.net** space: Definitions *D*, *MyD* and *E* and objects of types *A*, *B*, *C*, X and *Y* respectively. The first three objects in turn implement the following sets of definitions: *{ D }*, *{ D, E }* and *{ MyD, E }* respectively, where *MyD* is a refinement of *D*. The objects of types *X* and *Y* use the service objects through definitions exclusively. For example, client *Y* uses *A* through *D*, *B* through *E* and *C* through *E* and *MyD*. The corresponding notations are *D(A)*, *E(B)*, *E(C)* and *MyD(C)*. The figure also shows interoperability mappings: Classes like *D* produced by a foreign producer are mapped to definitions. Conversely, definitions like *MyD* and object types like *C* created by Active Oberon for **.net** are mapped to reusable classes.

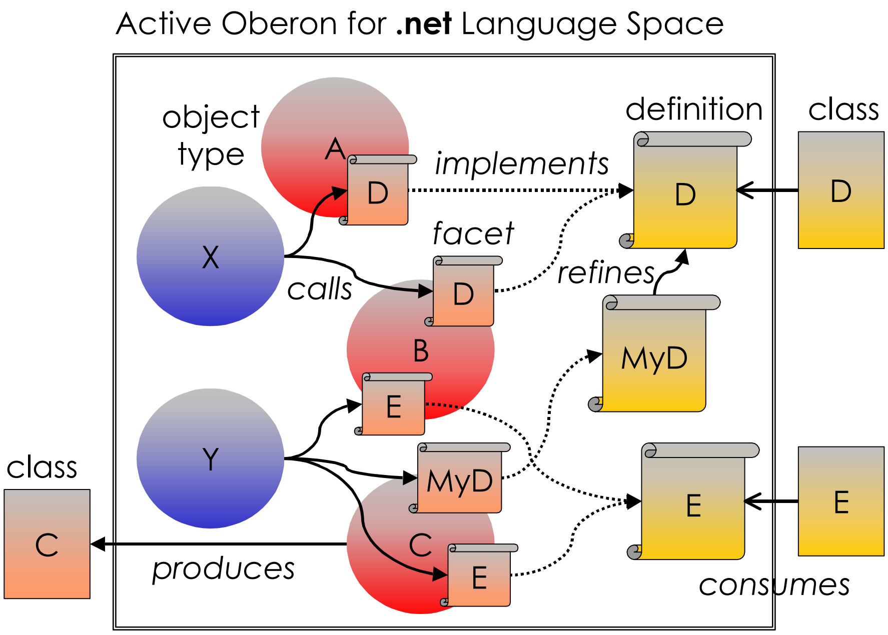

Figure 2. Active Oberon for .net's Object Architecture

5.3 Generic Objects and Polymorphism

Active Oberon for **.net** features a *generic* OBJECT construct, optionally followed by a set of names denoting *postulated interfaces*. This construct is used in polymorphic declarations and parameter lists as, for example in

```
VAR x: OBJECT; y: OBJECT { D };
PROCEDURE P (x: OBJECT; VAR y: OBJECT { D, E });
```

In both scenarios, *x* represents any object. In the first scenario, *y* stands for an object of any type implementing definition *D*. In the second scenario, *y* denotes an object variable that is guaranteed to implement definitions *D* and *E*.

The use of generically defined objects is supported by an *implementation test* statement. If the desired definition is a member of the list of postulated interfaces, it can safely be accessed directly. If, however, the desired definition is not postulated statically, its access needs to be safeguarded by an implementation test such as, for example,

```
IF x IMPLEMENTS D THEN … D(x).f(..); … END
```

All attempts of accessing unimplemented definitions result in a runtime error.

As a convenience, every object type is considered to automatically implement itself (interpreted as a definition). Therefore, if *x* is declared as a generic OBJECT and is currently referring to an object of type *Figure*, then *Figure(x).Display(X, Y)* would be a valid method call construct "narrowing" type OBJECT to type *Figure*.

5.4. Modules

Modules in Active Oberon for **.net** have the dual semantics of *namespace* and *singleton object*. In general, every module is both

- a *name-space* comprising a set of definitions and a set of object types, and
- an instance of the (anonymous) object type specified by the module's fields and methods (global variables and procedures)

The special cases of an empty set of definitions, an empty set of object types and an empty singleton object or any combination thereof are allowed. They correspond to special kinds of modules. For example, a module containing nothing but definitions could be viewed as a *definition module*.

Figure 3 exemplifies the dual role of modules.

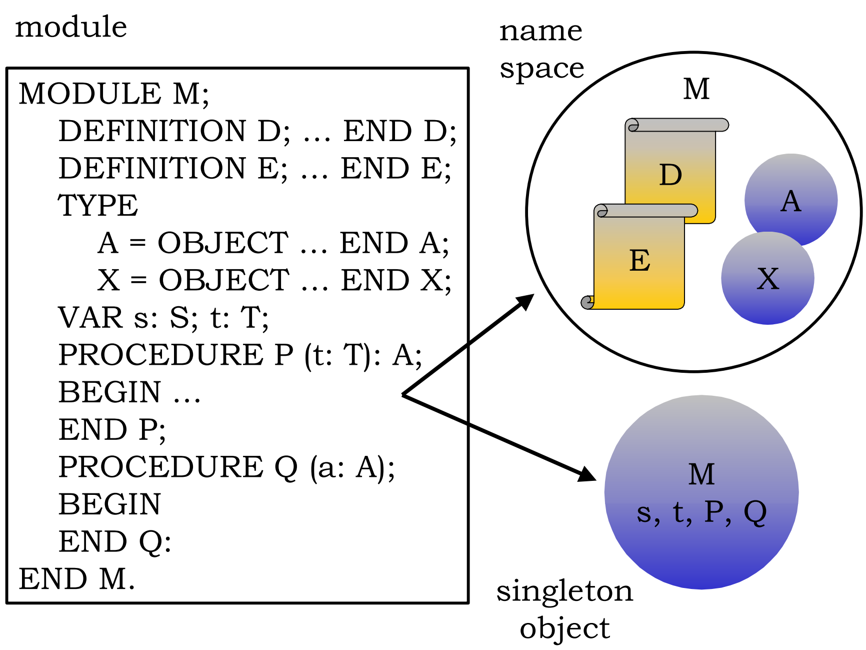

Figure 3. Active Oberon for .net's Module Concept

Notice that singleton objects defined by modules are not accessible via reference. Consequently, they can neither be instantiated explicitly nor can they be passed as parameters. In a sense, they are “second class citizens”.

Every module defines a *public view* consisting of the set of specially marked *exported* items. (Lists of) exported variables are by default *internal* that is the scope of their visibility is restricted to the embodying module. Public items have to be marked by either an { EXPORTED } modifier or by a *star-symbol* as in Oberon. An additional modifier is { PRIVATE }. In this case, the visibility is restricted to the embodying object declaration.

For example, the following declaration (where T denotes any type) defines (a) a name space *M* containing one exported definition *D* itself exporting variable *x* and method *h*, two exported object types *A* and *C* and (b) a singleton object *M* with exported fields *a*, and *t*, and a public method *F*. Definition *E* and object type *B* are *private* items. Variable *b* and procedure *P* are internal and invisible from the outside.

Both the exported contents of the name-space and the exported member elements of the singleton object are referred to from the outside by qualification: *M.D, M.A*, *M.t*, *M.F* etc.

```
MODULE M;

  DEFINITION { EXPORTED } D;
    VAR x*: X; y: Y;

    PROCEDURE f (..);
      VAR u, v: U;
    BEGIN ..
    END f;

    PROCEDURE { PRIVATE } g (..): ..;
    PROCEDURE { EXPORTED } h (..);
  END D;

  DEFINITION { PRIVATE } E;
    VAR t: T;

    PROCEDURE p (..);
      VAR a, b: A;
    BEGIN ..
    END p;

    PROCEDURE q (..): ..;
  END E;

  TYPE
    A* = OBJECT
      VAR { EXPORTED } u: U;
      PROCEDURE p* (..);
      BEGIN
      END P;
    END A;

   TYPE { PRIVATE }
```

```
    B = OBJECT .. END B;
    C* = OBJECT .. END C;

  VAR a*: A; b: B; t*: T;

  PROCEDURE P (..);
  BEGIN ..
  END P;

  PROCEDURE { EXPORTED } F (..): T;
  BEGIN ..
  END F;

BEGIN ..
END M;
```

Each module implicitly defines a namespace. Module names can therefore be hierarchically qualified. Example: *A.B.M*. The IMPORT clause allows import of items from foreign namespaces. If module *N* imports module *M*, then *N* is qualified to use (a) *M*'s exported ("*"-marked) definitions and object types at compile time, and (b) the singleton object *M*'s exported items at runtime.

In concluding this section we emphasize that modules are the only packaging entities in Active Oberon for **.net**. In particular, Active Oberon for **.net** neither supports explicit namespaces nor explicit assemblies.

5.5 Block Statement

A *block statement* is used by Active Oberon for **.net** to group a set of statements sharing certain processing properties. Its most general form is

```
BEGIN { modifiers }
  (* statement sequence 1 *)
ON EXCEPTION
  (* statement sequence 2 *)
END;
```

where both the *modifiers* part and the ON EXCEPTION part are optional.

A block statement is syntactically allowed whereever other statements are. It also serves as unified representation of the body of all scoped constructs like MODULE, PROCEDURE, and OBJECT.

The informal semantics of the general block statement is this: "Under control of the modifiers, execute statement sequence 1. In case of any exception while executing statement sequence 1, sequentially execute statement sequence 2".

The set of modifiers is conceptually open. Currently defined are SEQUENTIAL (sequential execution required), CONCURRENT (order-independent execution permitted), ACTIVE (execution under control of separate thread) and EXCLUSIVE (mutually exclusive execution within object scope). In case of omitted pragma specification, the system defaults to SEQUENTIAL.

6. Implementation Aspects

6.1. From the .net Platform View

The Active Oberon for **.net** language was carefully modeled together with its mapping to the **.net** platform. Here is a summary of the mapping:

module → assembly, name space, final class
definition → interface, abstract class
object type → sealed class
type OBJECT → class System.Object
implements → implements, inherits from (in the case of foreign classes)
INTEGER → System.Int32

However, still a few restrictions have to be accepted and some technical fine points have to be considered when plugging-in Oberon into **.net**. Here are some examples worth mentioning:

- **Delegates.** We extended Oberon's concept of procedure variables to *method variables*. While procedure variables are restricted to global procedure values, method variables may take arbitrary method values. No language changes were necessary. The compiler now simply allocates a pair of pointers *(code pointer, object base)*, for every method variable.
- **Overloading.** We extended Oberon to support method overloading. In the case of ambiguity, we require the programmer of a call 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.
- **Supercalls.** Because Oberon does not support subclassing, no special construct is provided for supercalls. However, calls of methods of imported superclasses (appearing as definitions) are possible by simply using fully qualified method names.
- **Assemblies**. *IMPORT* clauses have been extended by a compiler hint *{ "AssemblyName" }* for the specification of the assembly containing the class to be imported. If no hint is given, the assembly name defaults to the name of the imported namespace or, in case of a name beginning with *System*, to *mscorlib*.

In its current state the implementation treats *active objects* as independently processing entities, where a built-in method called *SETPRIORITY* is used to set an object's processing priority. In a next stage we shall implement synchronization primitives for (a) the protection of shared resources from competitive accesses and (b) the support of assertion-based synchronization among of collaborating objects. Our goal is a closest possible port of Active Oberon's *EXCLUSIVE* option and of its *AWAIT predicate* statement to **.net**.

In other cases, Active Oberon for **.net** does not exploit the potential of the **.net** language platform. For example, Oberon internally does not discern any difference between the notions of assembly, module, and namespace. Essentially, these notions are simply identified with the module construct. Also, Oberon does not currently make use of the powerful versioning support provided by **.net**.

Another implementation issue worth noting is the **.net** *reflection API*. We have made intense use of this API to internalize imported classes and interfaces at compilation time and to generate readable interfaces in Oberon syntax. We shall need to make further use of reflection by shortcutting the IL-assembler via a code generator that uses the reflection-emit API.

6.2. From the Compiler View

The current version of the Active Oberon for **.net** compiler uses a simple and fast one-pass, recursive descent strategy. Following our tradition, we have consciously applied the "80-20 rule" in the construction of our compiler, and thereby left unimplemented any excessively expensive but rarely occurring case or combination of cases. As a consequence, some *implementation restrictions* imposed by the compiler have to be accepted.

Also, in some cases, the compiler relies on explicit annotations of the form *{ directive }* provided by the programmer. However, such directives are not considered as part of the Active Oberon for **.net** language.

Implementation Restrictions

- **Nesting of declarations.** The current version of the compiler does neither allow nested definitions nor nested object types nor nested procedures.
- **Declaration of active objects.** Active objects (with a body part) must be annotated by an { ACTIVE } directive following the OBJECT keyword.
- **Forward declarations.** In principle, the compiler relies on the principle of "declare before use" (in the program text). However, this rule cannot be complied with in cases of mutual use as, for example, in the cases of object types mutually referring to each other and procedures mutually calling each other. While the compiler is able to handle the former case automatically, a forward procedure declaration in the form of a stand-alone procedure signature is needed in the second case.

7. Project History

The Active Oberon for **.net** project started in the summer 1999 and is supposed to end in summer 2002. In August 1999, a first version of a Active Oberon for **.net** compiler existed. It was a stand-alone cross compiler running under Native Oberon and producing symbolic IL assembler code. By February 2000, the compiler has been ported to the **.net** platform, and it now supports language interoperability. In particular, the current compiler is capable to re-use an existing class library and to produce reusable classes. It employs a simple command line interface and assumes source code in the form of plain text. It has recently passed the self-compilation test.

In a next major step we shall develop a new version of the compiler based on an explicit data structure that connects the front-end and back-end parts. The release of this compiler is planned for end of 2001. It will in particular feature (a) complete support of data types requested by the common language subset, (b) complete support of the definition construct, (c) directly generated IL code via reflection API, (c) synchronization constructs for active objects. Also planned for the nearer future are a few substantial demo-applications using the *WinForms* and *WebForms* (ASP+) API, a ”*light”* implementation of XMLDOM in Active Oberon for .net, an implementation of distributed Active Oberon for **.net** objects under SOAP control, and a more fine-grained integration of Active Oberon for **.net** into Visual Studio.

8. Conclusion

The implementation of Active Oberon for **.net** has reached an operational state. Thanks to the powerful **.net** basis and the careful design of the Active Oberon for **.net**’s object architecture, no substantial problems have occurred so far. First demo programs implemented in Active Oberon for **.net** are promising in every respect. We are confident that Active Oberon for **.net** may become an interesting choice of implementation language in a multilingual world.

We finally hope that Active Oberon’s active object model in combination with its unified and symmetric concept of abstraction will take influence on the mindset of software architects and in the end lead to a new generation of less complex and truly component-oriented “scalable” software systems.

9. Acknowledgement

My thanks go to James Plamondon and Pierre-Yves Saintoyant for inviting us to participate in Project 7, and to Brad Merrill, Dan Fay, Jim Miller, Erik Meijer, and Don Syme for managing it so effectively. My thanks also go to my former and current collaborators Thomas Frey, Philipp Kramer, Hanspeter Högger, Ben Smith-Mannschott, and Patrick Reali for their support, great contributions, and constructive critisism. Last but not least I gratefully acknowledge Brian Kirk’s most valuable comments after reviewing an earlier version of this paper.

Appendix A: Active Oberon for .net Syntax

module = MODULE ident ";" [ImportList] { definition } ObjectSpec.
ImportList = IMPORT import { "," import } ";" .
import = ["{" string "}"] qualident.
definition = DEFINITION ident [REFINES qualident] ";" declarations END ident ";".
declarations = { CONST { ConstantDeclaration ";" } | TYPE { TypeDeclaration ";" } | VAR { VariableDeclaration } | ProcedureDeclaration }.
ConstantDeclaration = identpub "=" ConstExpression.
VariableDeclaration = IdentList ":" type.
ConstExpression = expression.
TypeDeclaration = identpub "=" type.
type = qualident | ArrayType | RecordType | ObjectType | EnumType | ProcedureType.
ArrayType = ARRAY length { "," length } OF type.
length = ConstExpression | "*".
RecordType = RECORD FieldListSequence END.
FieldListSequence = FieldList { ";" FieldList }.
FieldList = [IdentList ":" type].
IdentList = identpub { "," identpub }.
ObjectType = OBJECT [[IMPLEMENTS qualident { "," qualident }] ObjectSpec].
ObjectSpec = declarations (BlockStatement | END) ident.
EnumType = "(" IdentList ")".
ProcedureType = PROCEDURE [FormalParameters].
ProcedureDeclaration = ProcedureHeading ";" [ProcedureBody ident].
ProcedureHeading = PROCEDURE ["^" | "&"] identpub [FormalParameters].
ProcedureBody = declarations BlockStatement.
FormalParameters = "(" [FPSection { ";" FPSection }] ")" [":" qualident].
FPSection = [VAR] ident { "," ident } ":" FormalType.
FormalType = { ARRAY OF } qualident.
StatementSequence = statement { ";" statement }.
statement = [assignment | ProcedureCall | IfStatement | CaseStatement | WhileStatement | RepeatStatement | LoopStatement | ForStatement | EXIT | RETURN [expression] | BlockStatement].
assignment = designator ":=" expression.
ProcedureCall = designator ["{" signature "}"] [ActualParameters].
ActualParameters = "(" [ExpList] ")" .
signature = "(" [FTSection { ";" FTSection }] ")" [":" qualident].
FTSection = [VAR] FormalType { "," FormalType }.
IfStatement = IF expression THEN StatementSequence { ELSIF expression THEN StatementSequence } [ELSE StatementSequence] END.
CaseStatement = CASE expression OF case { "|" case } [ELSE StatementSequence] END.
case = [CaseLabelList ":" StatementSequence].
CaseLabelList = CaseLabels { "," CaseLabels }.
CaseLabels = ConstExpression [".." ConstExpression].
WhileStatement = WHILE expression DO StatementSequence END.
RepeatStatement = REPEAT StatementSequence UNTIL expression.
LoopStatement = LOOP StatementSequence END.

ForStatement = FOR ident “:=” expression TO expression [BY ConstExpression] DO StatementSequence END .
ExpList = expression { "," expression }.
expression = SimpleExpression [relation SimpleExpression].
relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IMPLEMENTS.
SimpleExpression = ["+"|"-"] term {AddOperator term}.
AddOperator = "+" | "-" | OR .
term = factor {MulOperator factor}.
MulOperator = "*" | "/" | DIV | MOD | "&" .
factor = number | CharConstant | string | NIL | set | designator [“{“ signature “}”] [ActualParameters] | "(" expression ")" | "~" factor.
set = "{" [element { "," element }] "}".
element = expression [".." expression].
designator = qualident { "." ident | "[" ExpList "]" | "(" qualident ")" }.
ident = letter { letter | digit | "_" | "$" }.
number = integer | real.
integer = digit { digit } | digit { hexDigit } "H" .
real = digit { digit } "." { digit } [ScaleFactor].
ScaleFactor = ("E" | "D") ["+" | "-"] digit { digit }.
hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
letter = "A" | ... "Z" | "a" | ... "z".
CharConstant = """ character """ | digit { hexDigit } "X".
string = """ { character } """ .
qualident = [ident "."] ident.
identpub = ident ["*" | “#”].

Appendix B: Built-In Procedures and Functions

ABS(x)	absolute value
ODD(x)	x MOD 2 = 1
CAP(x)	corresponding capital letter
LEN(v, n)	length of v in dimension n
LEN(v)	is equivalent with LEN(v, 0)
MAX(T)	maximum value of type T, maximum element of sets
MIN(T)	minimum value of type T
ORD(x)	ordinal number of x
CHR(x)	character with ordinal number x
SHORT(x)	x as next shorter type (truncation possible)
LONG(x)	x as next longer type
ENTIER(x)	largest integer not greater than x. Note that ENTIER(i/j) = i DIV j
INC(v)	v := v+1
INC(v, x)	v := v+x
DEC(v)	v := v-1
DEC(v, x)	v := v-x
INCL(v, x)	v := v + {x}
EXCL(v, x)	v := v - {x}
NEW(v, ...)	allocate v
HALT(x)	terminate program execution
SETPRIORITY(p)	set priority of object-thread

Appendix C: Sample Programs

1 Puzzle

This program simulates the well-known puzzle depicted below. It appears as a regularly square-tiled rectangle with one hole. By repetitively sliding neighboring tiles into the hole, any configuration can finally be reached. The program uses the *WinForms* API.

```
MODULE Puzzle;

IMPORT System, {"System.Drawing"}System.Drawing,
{"System.WinForms"}System.WinForms;

TYPE
  MyForm = OBJECT IMPLEMENTS System.WinForms.Form;
    CONST
      timerDelay = 500; (* ms *)
      Delay = 1000;
      left = 0; right = 1; up = 2; down = 3;
    VAR
      nofElems, nofSubDiv: INTEGER;
      elemWidth, elemHeight: INTEGER;
      mainImage: System.Drawing.Image;
      Timer: System.Threading.Timer;
      Random: System.Random;
      pos: ARRAY * OF INTEGER;
      hole, prevDir: INTEGER;
      pboxes: ARRAY * OF System.WinForms.PictureBox;
      locked: BOOLEAN;

      PROCEDURE Crop(source: System.Drawing.Image;
        r: System.Drawing.Rectangle): System.Drawing.Bitmap;
      VAR
        g: System.Drawing.Graphics;
        target: System.Drawing.Bitmap;
        s: System.Drawing.Rectangle;
      BEGIN
        NEW(target, r.get_Width(), r.get_Height());
        s := System.Drawing.Rectangle.FromLTRB(0, 0,
             r.get_Width(), r.get_Height());
        g := System.Drawing.Graphics.FromImage
             {System.Drawing.Image): System.Drawing.Graphics}(target);
        g.DrawImage(source, s, r, System.Drawing.GraphicsUnit.Pixel);
        g.Dispose();
        RETURN target;
      END Crop;

      PROCEDURE MoveElem(pbox: System.WinForms.PictureBox;
        dx, dy, nofSteps: INTEGER);
```

```
VAR
  i, j, top, left: INTEGER;
  loc: System.Drawing.Point;
BEGIN
  left := pbox.get_Left(); top := pbox.get_Top();
  loc.set_X(left); loc.set_Y(top); pbox.set_Location(loc);
  FOR i := 1 TO nofSteps DO
    FOR j := 1 TO Delay DO END;
    loc.set_X(left + i * dx DIV nofSteps);
    loc.set_Y(top + i * dy DIV nofSteps);
    pbox.set_Location(loc)
  END
END MoveElem;

PROCEDURE TimerEventHandler(state:OBJECT);
VAR
  dir, avoid, tmp, holeDst, dx, dy: INTEGER;
BEGIN
  IF ~locked THEN
    locked := TRUE;

    (* move hole *)
    REPEAT
      dir := Random.Next(4);
      CASE dir OF
        left: IF hole MOD nofSubDiv = 0 THEN dir := right END
        | right:
          IF (hole+1) MOD nofSubDiv = 0 THEN dir := left END
        | up: IF hole < nofSubDiv THEN dir := down END
        | down:
          IF hole >= nofElems - nofSubDiv THEN dir := up END
      END;
      dx := 0; dy := 0;
      CASE dir OF
        left: dx := -1; avoid := right
        | right: dx := 1; avoid := left
        | up: dy := -1; avoid := down
        | down: dy := 1; avoid := up
      END
    UNTIL avoid # prevDir;
    prevDir := dir;

    holeDst := hole + dx + dy * nofSubDiv;
    MoveElem(pboxes[pos[holeDst]],
      -dx * elemWidth, -dy * elemHeight, 100);
    tmp := pos[hole]; pos[hole] := pos[holeDst];
    pos[holeDst] := tmp;
    hole := holeDst;
    locked := FALSE
  END
END TimerEventHandler;
```

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

PROCEDURE InitPositions();
VAR
  i, a, b, t: INTEGER;
BEGIN
  NEW(pos, nofElems);

  (* random permutations *)
  FOR i := 0 TO nofElems-1 DO pos[i] := i END;
  FOR i := 0 TO nofElems-1 DO
    a := Random.Next(nofElems); b := Random.Next(nofElems);
    t := pos[a]; pos[a] := pos[b]; pos[b] := t
  END;
  hole := 0; prevDir := -1;
END InitPositions;

PROCEDURE InitElems();
VAR
  i, x, y: INTEGER;
  pbox: System.WinForms.PictureBox;
  bounds, cropRect: System.Drawing.Rectangle;
  ctls: System.WinForms.Control$ControlCollection;
BEGIN
  NEW(pboxes, nofElems);
  ctls := System.WinForms.Form.get_Controls();
  FOR i := 0 TO nofElems-1 DO
    IF i = hole THEN pbox := NIL
    ELSE
      NEW(pbox);
      x := (i MOD nofSubDiv) * elemWidth;
      y := (i DIV nofSubDiv) * elemHeight;
      bounds := System.Drawing.Rectangle.FromLTRB(x, y,
        x + elemWidth, y + elemHeight);
      pbox.set_Bounds(bounds);
      x := (pos[i] MOD nofSubDiv) * elemWidth;
      y := (pos[i] DIV nofSubDiv) * elemHeight;
      cropRect := System.Drawing.Rectangle.FromLTRB(x, y,
        x + elemWidth, y + elemHeight);
      pbox.set_Image(System.Drawing.Image(
        Crop(mainImage, cropRect)));
      ctls.Add(System.WinForms.Control(pbox))
    END;
    pboxes[pos[i]] := pbox
  END
END InitElems;
```

```
    PROCEDURE NEW(image: System.Drawing.Image;
      subDivisions: INTEGER);
    VAR
      MouseHandler:System.WinForms.MouseEventHandler;
      TimerHandler:System.Threading.TimerCallback;
      x, y: INTEGER;
      sz: System.Drawing.Size;
    BEGIN
      NEW(Random);
      mainImage := image;
      nofSubDiv := subDivisions; nofElems := nofSubDiv * nofSubDiv;
      elemWidth := image.get_Width() DIV subDivisions;
      elemHeight := image.get_Height() DIV subDivisions;
      InitPositions();
      InitElems();

      locked := FALSE;

      (* Hook the timer event of the Timer *)
      NEW(TimerHandler, TimerEventHandler);
      NEW(Timer, TimerHandler, NIL, 0, timerDelay);

      NEW(sz, subDivisions * elemWidth, subDivisions * elemHeight);
      System.WinForms.Form.set_ClientSize(sz)
    END NEW;
  END MyForm;

VAR
  inst:MyForm;
  nofSubDiv: INTEGER;
  fileName: System.String;

  PROCEDURE GetParams(VAR nofSubDiv: INTEGER;
    VAR imageFileName: System.String);
  VAR Params: ARRAY * OF System.String;
    i: INTEGER;
    s, fn: ARRAY * OF CHAR;
    ok: BOOLEAN;
  BEGIN
    Params:=System.Environment.GetCommandLineArgs();
    IF LEN(Params) > 1 THEN
      nofSubDiv :=
        System.Convert.ToInt32{System.String):INTEGER}(Params[1])
    ELSE nofSubDiv := 3
    END;
    IF LEN(Params) > 2 THEN imageFileName := Params[2]
    ELSE imageFileName := "checker.gif"
    END
  END GetParams;

BEGIN
```

```
  GetParams(nofSubDiv, fileName);
  NEW(inst, System.Drawing.Image.FromFile(fileName), nofSubDiv);
  System.WinForms.Application.Run(System.WinForms.Form(inst));
END Puzzle.
```

2 BunnyRace

This program simulates a bunny-race, where each bunny is mapped to an active Oberon object. It also makes use ofthe *WinForms* API.

```
MODULE BunnyRace; (* Thomas Frey *)
IMPORT System, {"System.WinForms"} System.WinForms, {"System.Drawing"}
System.Drawing;

CONST NofBunnies = 5;Width = 1000; Delay = 5000;
  VAR Bunny: System.Drawing.Image;

TYPE
  BunnyPanel = OBJECT { ACTIVE } IMPLEMENTS System.WinForms.Panel;
    VAR
      i: INTEGER;
      x, y: INTEGER;
      Frame: INTEGER;
      Walk: BOOLEAN;
      Alive: BOOLEAN;
      Number: INTEGER;

    PROCEDURE OnPaint(sender: OBJECT;
      pe: System.WinForms.PaintEventArgs);
      VAR g: System.Drawing.Graphics; sr, dr:System.Drawing.Rectangle;
    BEGIN
      g := pe.get_Graphics();
      NEW(sr, Frame*135, 0, 135, 129);
      NEW(dr, x-135, y, 135, 129);
     g.DrawImage(Bunny, dr, sr, System.Drawing.GraphicsUnit.Pixel)
    END OnPaint;

    PROCEDURE NEW(Nr:INTEGER);
    VAR PaintHandler:System.WinForms.PaintEventHandler;
    BEGIN
      Alive:=TRUE; x:=135; Number := Nr;
      (* Hook the PaintHandler *)
      NEW(PaintHandler, OnPaint);
      System.WinForms.Panel.AddOnPaint(PaintHandler)
    END NEW;

    PROCEDURE Hops();
    BEGIN
      WHILE Alive DO i:=0;
        WHILE i < Delay DO INC(i) END;
```

```
      IF x>System.WinForms.Panel.get_Width() THEN Alive:=FALSE;
        WRITELN("Bunny_",Number," finished.")
      END;
      IF Walk THEN
        x:=(x + 6) MOD (System.WinForms.Panel.get_Width() + 50)
      END;
      Frame:=(Frame + 1) MOD 8;
      System.WinForms.Panel.Refresh();
    END
  END Hops;

BEGIN
  Hops();
END BunnyPanel;

WinRectsObj = OBJECT IMPLEMENTS System.WinForms.Form;
  VAR
    Bunnies: ARRAY OF BunnyPanel;
    Start: System.WinForms.Button;

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

  PROCEDURE Button_Click(sender:OBJECT; e:System.EventArgs);
  VAR i:INTEGER;
  BEGIN i:=0;
    WHILE i < NofBunnies DO
      Bunnies[i].Walk:=TRUE; INC(i)
    END
  END Button_Click;

  PROCEDURE NEW();
  VAR p:System.Drawing.Point;
    s:System.Drawing.Size;
    i:INTEGER;
    Random: System.Random;
    handler: System.EventHandler;
  BEGIN
    NEW(Random);
    System.WinForms.Form.set_Text("Bunny Thread Race");
    NEW(Start);
    NEW(p, 0, NofBunnies*129); Start.set_Location(p);
    NEW(s, Width, 25); Start.set_Size(s);
    Start.set_Text("Go!!!");
    System.WinForms.Form.get_Controls()
      .Add(System.WinForms.Control(Start));
    NEW(handler, Button_Click);
    Start.AddOnClick(handler);
    NEW(s, Width, NofBunnies*129+25);
    System.WinForms.Form.set_ClientSize(s);
```

```
      NEW(Bunnies, NofBunnies);
      i:=0;
      WHILE i < NofBunnies DO
        NEW(Bunnies[i], i);
        System.WinForms.Form.get_Controls()
          .Add(System.WinForms.Control(Bunnies[i]));
        NEW(p, 0, i*129);
        System.WinForms.Panel(Bunnies[i]).set_Location(p);
        NEW(s, Width, 129);
        System.WinForms.Panel(Bunnies[i]).set_Size(s);
        INC(i)
      END
    END NEW;
  END WinRectsObj;

VAR inst: WinRectsObj;

BEGIN
  Bunny := System.Drawing.Image.FromFile("BunnyLinear.gif");
  WRITELN("Bunny Thread Race");
  WRITELN("===================");
  WRITELN("In Active Oberon for .net");
  NEW(inst);
  System.WinForms.Application.Run(System.WinForms.Form(inst))
END BunnyRace.
```