Concurrent object-oriented programming on .NET

P. Nienaltowski, V. Arslan and B. Meyer

Abstract: The **SCOOP** model (Simple concurrent object-oriented programming) offers a comprehensive approach to building high-quality concurrent and distributed systems. The model takes advantage of the inherent concurrency implicit in object-oriented programming to provide programmers with a simple extension enabling them to produce concurrent applications with little more effort than sequential ones. In the paper, the authors discuss the basic concepts of the model, such as *processors* and *separate objects*. They also present SCOOPLI, a library implementation of SCOOP for the .NET platform. They show how SCOOP concepts are mapped to .NET constructs, and discuss distributed programming with SCOOPLI, with a focus on .NET Remoting capabilities. Several programming examples illustrate the discussion.

1 Introduction

Concurrent programming in its many variants from multithreading to multiprocessing, distributed computing, Internet applications and Web services, has become a required component of ever more types of systems, including some that were traditionally thought of as essentially sequential. However, the industry is still looking for a good way to produce concurrent applications. The contrast with sequential programming is stark: there, a widely accepted set of ideas (standard control and data structuring techniques, modularity and information hiding, object-oriented principles) have displaced the lower-level concepts that used to predominate, but the techniques commonly used to produce concurrent applications are elementary and often haphazard. Many developers, for example, want to add multithreading to their systems, but the proper use of multithreading remains a black art. The continuing discussions about how multithreading breaks the Java Memory Model is a typical symptom of this situation. The software field badly needs a concurrent programming technique enjoying the same simplicity and inspiring the same confidence as the accepted constructs of sequential programming.

The **SCOOP** model is an attempt to provide this simple basis. SCOOP [1, 2: Chap. 31] takes object-oriented programming as given, in a form based on the concepts of Design by contract, and extends them in a minimal way (one language keyword and a few library mechanisms) to cover concurrency and distribution. To address such requirements of concurrent processing as mutual exclusion, synchronisation and wait conditions, SCOOP gives new semantics to well known constructs (argument passing, preconditions) where the standard sequential semantics could not be applied anyway. The model is applicable to many different physical setups, from multiprocessing to multithreading, network programming, Web services and distributed computation. It takes advantage of the inherent concurrency implicit in object-oriented programming to shield programmers from low-level concepts such as semaphores, letting them instead produce concurrent applications along the same lines as sequential ones.

SCOOP has only had prototype implementations so far. Along with continuing to refine the model, we are building a production implementation. While intending to support many different platforms and concurrency mechanisms thanks to the model's generality (Fig. 1), we have chosen Microsoft .NET as our first implementation and reference platform, taking advantage of specific .NET mechanisms — provided in particular by the *System.Runtime.Remoting* and *System.Threading* namespaces — which constitute an excellent basis for SCOOP. For generality the focus of the implementation is a language-independent library, described in this paper: SCOOPLI: a library-based implementation of SCOOP for .NET. Thanks to the multilanguage character of .NET, developers using different languages can take advantage of the concurrency mechanism provided by SCOOPLI.

2 SCOOP model

SCOOP stands for *simple concurrent object-oriented programming*. The extension covering full-fledged concurrency and distribution adds a single new keyword to the Eiffel programming language — **separate**.

2.1 Generality

SCOOP has a two-level architecture (see Fig. 1). The top layer of the mechanism is platform-independent. To perform concurrent computations, applications use the 'separate' mechanism implemented at that level. Internally, the implementation relies on some concrete concurrent architecture, made accessible through one of the implementation libraries in the bottom layer. Possibilities listed in Fig. 1 include:

- an implementation using the .NET platform, especially its Remoting and Threading mechanisms, described in this paper
- a thread-based implementation, e.g. with POSIX threads
- a multithreading implementation on a real-time operating system, e.g. Windows CE .NET with the .NET Compact Framework.

IEE Proceedings online no. 20030992

doi: 10.1049/ip-sen:20030992

Paper received 29th June 2003

The authors are with the chair of Software Engineering, Swiss Federal Institute of Technology (ETH), 8092 Zurich, Switzerland

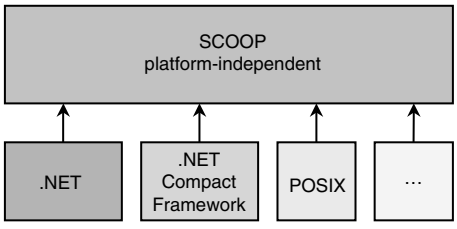

Fig. 1 *Two-level architecture of SCOOP*

The architecture simplifies the task of application developers, who only need to use the general concurrency mechanism implemented in the top layer; the bottom-layer libraries provide the binding to the actual concurrency platform [3].

2.2 Processors

In object-oriented computation, the basic mechanism is a feature call such as *x.f*(*a*), with the following semantics (Fig. 2): the client object calls feature *f* on the supplier object attached to *x*, with argument *a*. In a sequential setting, such calls are *synchronous*, blocking the client until the supplier has terminated the execution of the feature.

To support concurrency, SCOOP allows the use of more than one *processor* to handle execution of features. Every object has a *handler*: a processor in charge of executing feature calls on the object. If the client and supplier objects in a feature call have different handlers, the call becomes asynchronous: in Fig. 2, the computation on *Object 1* can move ahead without waiting for the call on *Object 2* to terminate.

Processors are the principal new concept for adding concurrency to the framework of sequential object-oriented computation; while a sequential system is limited to one processor, a concurrent system may have any number of processors. A processor is an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects. It does not have to be associated with a physical processor; it may also be implemented by a process of the operating system, or a thread in a multithreading environment. In the .NET Framework, processors can be mapped to *application domains* (see Section 4).

Viewed by the software, a processor is an abstract concept; the same concurrent application may be executed on very different architectures without any change to its source text (see Section 3).

2.3 Separate calls

Since the effect of a call depends on whether the client and the supplier objects are handled by the same processor or by different ones, the software text must distinguish unambiguously between these two cases. For declarations of variables or functions, normally appearing as *x*: *SOME_CLASS*, a new form is now possible, *x*: **separate** *SOME_ CLASS*.The keyword **separate** indicates that *x* is handled by a different processor, so that calls on *x* will be asynchronous. With such a declaration, any creation instruction **create** *x.make*(...) will use a new processor to handle calls on *x*. The declaration does not specify which processor to use for handling the object. What matters is that this processor is different from the processor handling the current object [Note 1].

Rather than an individual variable or function, it is also possible to declare a class as separate, as in **separate class** *SOME_CLASS* (instead of the usual class *OTHER_CLASS*). Then any variable of the corresponding type will be treated as separate. *SOME_CLASS* will be called a *separate class* [Note 2], and all its instances will be *separate objects*. If a target of a call is a separate expression, i.e. a separate entity or an expression involving at least one separate entity, such call is referred to as *separate call*. In Fig. 2, *x* is a separate entity, *Object 2* is a separate object and *x.f*(*a*) is a separate call.

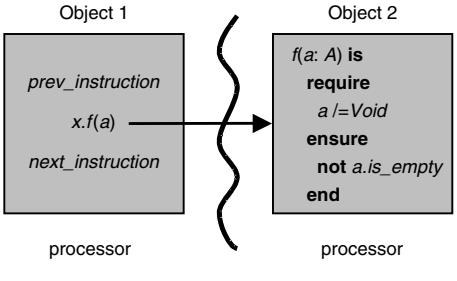

Fig. 2 *Feature call in SCOOP*

2.4 Synchronisation and communication

SCOOP addresses the synchronisation and communication needs of concurrent programming, including mutual exclusion, locking and waiting, through argument passing and design by contract.

2.4.1 Argument passing:

A basic rule of SCOOP is that a separate call on target *x*, *x.f* (...), where *x* is separate, is only permitted if *x* itself is an argument of the enclosing routine, and that calling a routine with such a separate argument will cause waiting until the corresponding separate object is exclusively available to the caller. So if you call *r* (*a*), or *y.r* (*a*), with

r (*x*: **separate** *SOME_TYPE*) **is** ...

the call will wait until no other client is using *a* in this way. This rule provides the basic synchronisation mechanism. It also avoids a common mistake of concurrent programming: to misinterpret that in two successive calls on a separate object, for example

```
that_stack.push (some_value)
...
x := that_stack.top
```

by assuming that nothing may have happened to the object in between — so that in the example the object retrieved is the object previously stored at the top of the stack. In a concurrent setting, any other clients may interfere with the object between the two calls. Under SCOOP, both calls using *that_stack* as a target must be in routines of which *that_stack* is an argument. If these are different routines, no confusion is likely; if they are the same routine, the rule guarantees that the routine will hold the object for the entire duration of every call, to the exclusion of any other clients.

2.4.2 Preconditions:

A routine may have a precondition and a postcondition, as in

Note 1: Section 4.1 describes how processors are mapped to physical resources.

Note 2: Eiffel syntax ensures that a class may be at most one of: *separate*, *expanded* or *deferred* [2]. The separateness of a class is not inherited: a class is separate or not according to its own declaration, regardless of its parents' status.

```
put (buffer: separate BUFFER [G]; value: G) is
    -- Store value into buffer.
require
    buffer_not_full: not buffer.is_full
    value_provided: value /= Void
do
    buffer.put (value)
ensure
    buffer_not_empty: not buffer.is_empty
end
```

A precondition clause involving a call with a separate target, such as *buffer.is_full*, is called a *separate precondition*. The other clause appearing here, *value /= Void*, is not separate.

In sequential programs, preconditions are correctness requirements that the client object must fulfil before calling the routine on the supplier object. If one or more preconditions are not met, the client has broken the contract; for example, it has tried to store a value into a full buffer. Since the execution is sequential, the state of the buffer cannot change (no other client can try to consume an element from the buffer in the meantime).

In a concurrent context this does not apply any more; the buffer may be full when the client object is trying to store a value into it, but nothing prevents another client object from consuming an element from the buffer later on. A nonsatisfied separate precondition does not necessarily break the contract; it just forces the client object to wait until the precondition is satisfied.

This inapplicability in a concurrent context of the usual sequential interpretation of preconditions leads to the SCOOP use of separate preconditions: as *wait conditions* rather than correctness conditions. We have seen the basic synchronisation rule: in the case of a separate argument, any call will wait until the object is available. To obtain the full synchronisation mechanism, we add the rule that if the routine has a precondition clause using such a separate argument as target, for example **not** *buffer.is_full* — called a *separate precondition* — the call will wait until both:

- the object is available
- the separate precondition is satisfied.

The wait semantics only applies to separate preconditions. Others, such as *value /= Void*, retain their usual meaning as correctness conditions.

2.5 Example and comments on the scheduling policy

The following example (with postconditions omitted for brevity) applies these concepts to a producer–consumer scheme. Some objects are producing values and storing them into the shared buffer *buf*; others are consuming elements from that buffer. For both producers and consumers, *buf* is a separate object, declared as such in the source code of both classes. To perform any call to *buf*, a client (producer or consumer) must obtain an exclusive lock on *buf*. The SCOOP rule then implies embedding all the calls to *buf* in routines *store* and *consume_one*. Direct calls to *buf.put*, *buf.item* and *buf.remove* would be invalid.

```
class PRODUCER
feature
    store (buffer: separate BUFFER [G]; value: G) is
        -- Store value into buffer.
    require
        buffer_not_full: not buffer.is_full
        value_provided: value /= Void
    do
        buffer.put (value)
    end
random_gen: RANDOM_GENERATOR
buf: separate BUFFER [INTEGER]
produce_n(n: INTEGER) is
        -- Produce n integer values and store them into a
        buffer.
    local
        value: INTEGER
        i: INTEGER
    do
        from i := 1
        until i > n
        loop
            value := random_gen.next
            store (buf, value)
            -- buf.put (value) is forbidden here
            i := i + 1
        end
    end
end

class CONSUMER
feature
    consume_one (buffer: separate BUFFER [G]) is
        -- Consume one element from buffer.
    require
        buffer_specified: buffer /= Void
        buffer_not_empty: not buffer.is_empty
    do
        value := buffer.item
        buffer.remove
    end

buf: separate BUFFER [INTEGER]

consume_n (n: INTEGER) is
        -- Consume n elements from a buffer.
    local
        i: INTEGER
    do
        from i := 1
        until i > n
        loop
            consume_one (buf)
            -- buf.item and buf.remove are forbidden here
            i := i + 1
        end
    end
end
```

To call *consume_one* from routine *consume_n*, a consumer will pass *buf* as an argument. In the SCOOP access control policy, when one or more arguments of a routine are separate objects, the client must obtain exclusive locks on all these objects before executing the routine. Here the consumer object must obtain an exclusive lock on *buf* before executing *consume_one*. If another object is currently holding the lock, the client must wait until the lock has been released, then try to acquire it. The default policy is first-in, first-out. When the client succeeds in acquiring the lock:

- The separate precondition clauses are evaluated. If they all hold, the routine will execute, then release the lock.
- Otherwise, the object releases the lock and restarts the whole process from the beginning: acquiring the locks, then checking the separate precondition clauses. This allows other clients to access the supplier object and change its

state, so that the wait conditions required by our client may eventually be met.

The locking policy facilitates building correct concurrent programs and reasoning about them:

- No interference between client objects is possible since at most one client may hold a lock on a supplier object at any time. This helps find which object is responsible for possible breaches in the contract, such as breaking the supplier's invariant.
- The precondition rules ensure that correct calls do not violate the integrity of the supplier object.

2.6 Synchronisation and wait by necessity

Thanks to the asynchronous semantics of separate calls, a client executing separate calls is not blocked and can proceed with the rest of its computation. Later on, however, it may need to resynchronise with the supplier. Rather than introducing a specific language mechanism for this purpose, SCOOP relies on 'wait by necessity' [4], which causes the client to wait on the result of calls to *queries* (in particular functions), since it needs the result to proceed, whereas *commands* (procedures) do not require waiting. This mechanism is automatic and does not require programming the resynchronisation.

2.7 Interrupts

If a client is holding a lock on a supplier object for too long, at the detriment of another client judged more important, SCOOP allows interrupting the current holder through a call to the library routine *demand*. A successful call will cause an exception in the current holder, which must have accepted the possibility in advance, and handle the exception (usually by trying again later on). If the interrupt fails, the exception will happen in the 'challenger' object. This mechanism provides added flexibility without violating the requirements of design by contract and the ability to reason statically about programs and their correctness. It makes it possible in particular to program timeouts as illustrated by examples in [2].

3 SCOOPLI

The two-level architecture of SCOOP (see Fig. 1) suggests that the general concurrency mechanism (top layer) should be implemented in a platform-independent style. The key concepts at that level are *processor*, *separate object* and *separate call*. Only their mapping to platform-dependent constructs will differ from one platform to another. This section describes the top layer, SCOOP proper; the mapping of SCOOP concepts to .NET constructs will be considered in Section 4.

3.1 Library approach

We decided to begin the implementation of SCOOP with an Eiffel library rather than by extending the compiler. This provides several advantages, in particular the ability to 'play' with the model by trying out various refinements and extensions, and to implement it on several platforms without getting bogged down in compiler issues. The final implementation will be integrated as an extension to the Eiffel compiler.

3.2 Basic concepts

SCOOPLI relies on the concepts of *separate client* and *separate supplier*. The underlying basic notions 'client' and 'supplier' are taken in the following sense:

Let S *be a class. A class* C *which contains a declaration of the form* x: S *is said to be a* ***client*** *of* S. S *is then said to be a* ***supplier*** *of* C.

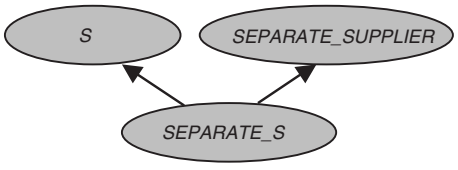

Fig. 3 *Use of multiple inheritance for declaration of separate entities*

Following this definition, a ***separate client*** is a class that contains a declaration of the form *x*: **separate** *S* [Note 3]. S is then said to be a ***separate supplier***. A separate client object is handled by a processor different from those of each of its separate suppliers [Note 4]. Therefore, any call of a feature on the separate supplier by the separate client object (*separate call*) is executed asynchronously, i.e. the separate client object can move to the next instruction without waiting for the current call to terminate.

The criteria for the design of the SCOOPLI interface were to make it as simple and easy to use as possible, and to maintain a clear correspondence with the SCOOP syntax.

3.3 Declaring a separate supplier

In SCOOP, one may declare separate supplier as either:

(a) *x*: **separate** *S*
(b) **separate class** *S* … **end**
x: *S*

SCOOPLI uses multiple inheritance to provide the same facility (see Fig. 3). All separate suppliers must inherit from the class *SEPARATE_SUPPLIER*:

```
class SEPARATE_S
inherit
   SEPARATE_SUPPLIER
   S
   ...
end
x: SEPARATE_S
```

3.4 Declaring a separate client

In SCOOP, there is no need to declare a class to be a separate client; any class can potentially become a separate client by using one or more separate entities (separate suppliers):

```
class MY_CLASS
feature
   x: separate S
   ...
end
```

SCOOPLI requires an explicit separate client declaration. Once again, the solution uses multiple inheritance: every separate client class must inherit from *SEPARATE_CLIENT*.

Note 3: This is expressed in the SCOOP syntax. The actual SCOOPLI interface is slightly different (see Section 3.4).

Note 4: In fact, SCOOP allows attaching a nonseparate object to a separate entity, so that both client and supplier objects are handled by the same processor. Our library does not allow such attachments.

```
class MY_CLASS -- separate client
inherit
   SEPARATE_CLIENT
feature
   x: SEPARATE_S -- separate supplier
   ...
end
```

3.5 Separate procedure calls

As noted, SCOOP requires that any call *x.f(a)*, where *x* is separate, must be embedded in a routine:

```
-- In class MY_CLASS
r (x1: separate S; a: SOME_CLASS) is
      -- Apply f to x1.
   do
      x1.f (a) -- here, a separate call is allowed
   end
...
r (x, a) -- here, a direct call to x.f(a) is prohibited
```

The routine may contain several separate calls to one or more separate suppliers, all accessed through formal arguments. As we have seen (Section 2.4), the locking mechanism obtains exclusive locks on all the corresponding objects prior to executing the routine. SCOOPLI follows these rules, with an appropriate interface:

```
-- In class MY_CLASS
r (x1: SEPARATE_S; a: SOME_CLASS) is
      -- Apply f to x1.
   do
      separate_routine (x1, agent x1.f (a))
      -- corresponds to x1.f (a)
   end
...
separate_execute ([x], agent r (x, a), Void)
         -- corresponds to r (x, a)
```

The calls *x.f* (*a*) and *r* (*x*, *a*) are wrapped in calls to *separate_routine* and *separate_execute*, respectively. Both routines are declared in the *SEPARATE_CLIENT* class. Let us have a closer look at them.

```
separate_routine (supplier: SEPARATE_SUPPLIER;
                  operation: PROCEDURE[])
```

Formal arguments:

- *supplier*

Denotes the separate supplier object on which the separate call to *operation* is made.

- *operation*

Denotes the routine to be called on the separate supplier object.

In the example above, *separate_routine* is called with arguments *x1* (for *supplier*) and **agent** *x1.f* (*a*) [Note 5] (for *operation*). Such a call correspond to *x.f* (*a*) in SCOOP.

```
separate_execute (requested_objects: TUPLE [];
                  action: PROCEDURE [];
                  wait_condition: FUNCTION [])
```

Formal arguments:

- *requested_objects*

Denotes the (tuple of) objects on which exclusive locks should be acquired before calling *action*.

- *action*

Denotes the routine to be called on the separate client object. *action* corresponds to the routine that 'wraps' separate calls.

- *wait_condition*

Denotes the Boolean function representing the wait condition [Note 6] for the call.

In the example, *separate_execute* is called with arguments [*x*] (for *requested_objects*), **agent** *r* (*x*, *a*) (for *action*), and *Void* (for *wait_condition*). Such a call corresponds to *r* (*x*, *a*) in SCOOP.

3.6 Wait conditions

In the example above there is no wait condition for routine *r*, since we assume that *r* has no precondition involving the separate object *x*. Should *r* have such a precondition, the part involving *x* would be extracted from the precondition and passed as *wait_condition* to *separate_execute*, e.g.

```
r (x1: SEPARATE_S; a: SOME_CLASS) is
   require
      x_not_empty: not x.is_empty
      a_positive: a > 0
   do
      separate_routine (x1, agent x1.f (a))
      -- corresponds to x1.f (a)
   end
...
r_wait_condition: BOOLEAN is
   do
      Result := not x.is_empty
   end
separate_execute ([x], agent r (x, a),
                  agent r_wait_condition)
-- corresponds to r (x, a)
```

3.7 Separate function calls

Direct application of features on separate supplier objects is prohibited (see Sections 2.3 and 3.5). This rule applies not only to procedures but also to functions.

If *some_value* is a function (of type *T*) defined in the class *SEPARATE_S*, and *x* is a separate supplier object of type *SEPARATE_S*, then every evaluation of *x.some_value* must be embedded in a routine that takes *x* as argument.

```
– in class MY_CLASS
y: T
...
r (x1: separate S) is
      -- Assign x1.some_value to y
   do
      y := x1.some_value
   end
...
r (x)
```

Note 5: **agent** *x.f* (*a*) is an object representing the operation *x.f* (*a*). Such objects, called *agents*, are used in Eiffel to 'wrap' routine calls [5]. One can think of agents as a more sophisticated form of .NET *delegates*.

Note 6: The wait condition is the part of a routine precondition that involves separate objects.

In SCOOPLI, calls to *x1.some_value* and *r* (*x*) are wrapped in calls to *separate_value* and *separate_execute*, respectively:

```
-- In class MY_CLASS
y: T
...
r (x1: SEPARATE_S) is
      -- Assign x1.some_value to y
   do
      y ?= separate_value (x1, agent x1.some_value)
   end
...
separate_execute ([x ], agent r (x), Void )
-- corresponds to r (x)
```

In the interface for *separate_value*:

```
separate_value (supplier: SEPARATE_SUPPLIER;
                function: FUNCTION []): ANY
```

the formal arguments have the following role:

- *supplier*

Denotes the separate supplier object on which the separate call to *function* is made.

- *function*

Denotes the function to be evaluated.

The return value is of type *ANY* (the most general type).

In the example, *separate_value* is called with arguments *x1* (for *supplier*) and **agent** *x1.separate_value* (for function).

The example uses an *assignment attempt* [Note 7] (?=, instead of standard assignment :=) because *separate_value* returns a result of type *ANY*, which we need as an object of type *T* (corresponding to the left-hand side of the assignment).

If the function returns an object of an *expanded type* [Note 8], a dedicated routine is used instead of *separate_value*, e.g. *separate_boolean_value* for *BOOLEAN*, *separate_integer_value* for *INTEGER*, etc. No assignment attempt is needed in such cases.

separate_execute is used in the same way as for separate procedure calls (see Section 3.5).

4 SCOOP on .NET

The previous descriptions are platform-agnostic. Our current implementation, as noted, targets .NET. This section describes how to map logical processors of the SCOOP model (Section 2.2) to .NET 'Appdomains', and how the implementation takes advantage of the multithreading model of the Microsoft .NET Framework.

4.1 Processes, application domains and threads on .NET

In most operating systems, processes provide isolation between several applications running on the same computer. In the .NET Framework a process consists of one or more *application domains* or Appdomains. Application domains can be considered as managed logical subprocesses. They provide isolation, unloading and security boundaries for managed .NET code. Using several application domains within a process increases server scalability [6]. Application domains can also be located on different computers.

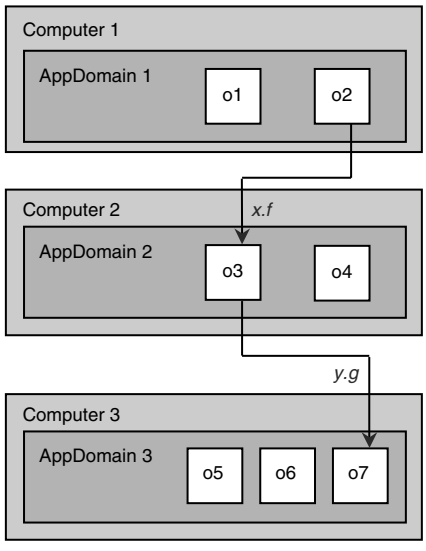

Fig. 4 *Distributed execution in SCOOPLI for .NET*

A .NET thread is a path of execution within an application domain. An application domain can have one or more threads, and any thread can be executed on different application domains at different times, since threads can cross application domain boundaries. But at any given time every thread is executed in one application domain. Cross-domain calls are allowed between application domains in one process as well as between application domains on different computers [7], thanks to the remoting capabilities of the .NET Framework [8].

4.2 Distributed execution

The basic rule of SCOOPLI for .NET, permitting distributed execution, is to use application domains as processors.

In the example of Fig. 4, the separate client object *o2*, located in *AppDomain 1* on *Computer 1*, calls *x.f*, where *x* is attached to the separate supplier object *o3*, which itself resides in *AppDomain 2* on *Computer 2*. As soon as the call *x.f* is initiated, *o2* can proceed without waiting for the termination of the call. Object *o3*, which now plays itself the role of a separate client object, calls *y.g*, where *y* is attached to the separate supplier object *o7*. Since *o7* resides in a different application domain located on a different computer than *o3*, call *y.g* has also separate (asynchronous) semantics.

Since processors are mapped to application domains, they can run as threads on the same machine, run on different machines, or use a combination of both solutions.

4.3 Specifying processor mapping

To preserve the generality of SCOOP programming, programs do not need to know about the precise mapping of processors to application domains and other physical resources. This is the role of a separate specification, the *Concurrency Control File* or CCF, such as the following example:

```
create
   local_nodes:
      system
         "Lincoln" (2): "c:\prog\appl1\appll.exe"
         "Roosevelt" (4): "c:\prog\appl2\appl2.dll"
         Current: "c:\prog\appl1\appl1.exe"
      end
   remote_nodes:
      system
         "Sinatra": "c:\prog\appl3\appl3.exe"
```

Note 7: An assignment attempt is similar to a dynamic cast, with the rule that if the assignment is impossible, the target receives the value Void.

Note 8: Expanded types, including the basic types *BOOLEAN*, *INTEGER*, *REAL*, *DOUBLE*, *CHAR*, and any other based on an *expanded class*, denote values rather than references [2].

```
        "Hemingway" (2): "c:\prog\appl4\appl4.exe"
      end
  end
  external
    Database_handler: "Warhol" port 9000
    ATM_handler: "Presley" port 8001
  end
  default
    port: 8001; instance: 10
  end
```

The *create* part specifies which physical resources to use for creating separate objects in instructions **create** *x.f*, where *x* is separate. The next two parts, called *local_nodes* and *remote_nodes*, deal with the mapping of processors to application domains. In the example above, the *local_nodes* entry specifies that:

- Two separate objects will be created in the application domain represented by the application *appl1.exe* on the computer *Lincoln*.
- The next four separate objects will be created in the application domain *appl2.dll* on the computer *Roosevelt*.
- The following ten will be created on the computer where the creation instruction is executed. The value 10 comes from the *instance* entry in the *default* part of the CCF.

For further separate object creations the allocation scheme is repeated, starting again with two separate objects on the computer *Lincoln*, four on *Roosevelt*, and so on.

We can also use application domains specified in *remote_nodes* and benefit from computers *Sinatra* and *Hemingway* to create separate objects. In the software text, we can choose between both groups by using a feature of the library class *CONCURRENCY* [1].

The *external* part specifies which physical resources to use for separate objects created outside the control of the program. In the example, we can get a reference to a separate database object from the computer *Warhol* on port 9000 by using a function

server (*name*: *STRING*; ...): **separate** *DATABASE*

with the argument *database_handler*.

The semantics of SCOOP and the compilation of a SCOOP or SCOOPLI application do not require a CCF; in its absence, a default scheme will determine the mapping of processors to application domains.

5 State of implementation and future work

The use of .NET Remoting has been a valuable asset to the current implementation of SCOOPLI, providing considerable simplification over the previous thread-based version.

The following features of SCOOP have been implemented so far:

- declaration and instantiation of separate objects
- call of procedures on separate objects
- exclusive locking of single separate objects
- argument passing (expanded and reference types)
- evaluation of functions implemented as routines
- evaluation of functions implemented as attributes
- assignment to nonseparate targets
- wait conditions
- wait by necessity.

The following mechanisms remain to be implemented:

- exclusive locking of multiple separate objects
- support for distributed execution
- CCF handler
- duel mechanism (interrupts).

The results achieved so far let us hope that the full implementation of SCOOP will provide the robust, trustable basis that will make concurrent programming as natural to programmers as its sequential variant.

6 Acknowledgments

The research work presented in this paper is part of the project 'SCOOP: Environment for dependable distributed and reliable object-oriented computing, based on the principles of Design by Contract'. We gratefully acknowledge the financial support of the Hasler Foundation (Berne, Switzerland). The project has also benefited from a Microsoft Rotor grant and is currently supported by a grant from the Swiss National Science Foundation.

7 References

1 Meyer, B.: 'Systematic concurrent object-oriented programming', *Commun. ACM*, 1993, **36**, (9), pp. 56–80
2 Meyer, B.: 'Object-oriented software construction' (Prentice Hall, Upper Saddle River, NJ, 1997, 2nd edn.)
3 Nienaltowski, P., and Arslan, V.: 'SCOOPLI: a library for concurrent object-oriented programming on .NET'. Presented at the 1st Int. Workshop on C# and .NET Technologies 2003, Plzen, Czech Republic, 5–7 February 2003
4 Caromel, D.: 'Towards a method of object-oriented concurrent programming', *Commun. ACM*, 1993, **36**, (9), pp. 90–102
5 Meyer, B.: 'Eiffel: the language' (Prentice Hall, 3rd edn.), to be published
6 'NET Framework SDK documentation' (Microsoft, 2002)
7 Dennis, A.: 'NET multithreading' (Manning, Greenwich, CT, 2003, 1st edn.)
8 Rammer, I.: 'Advanced .NET remoting' (Apress, Berkeley, CA, 2002, 1st edn.)