SCOOPLI: a library for concurrent object-oriented programming on .NET

Piotr Nienaltowski
Chair of Software Engineering
ETH Zurich
8092 Zurich, Switzerland
Piotr.Nienaltowski@inf.ethz.ch

Volkan Arslan
Chair of Software Engineering
ETH Zurich
8092 Zurich, Switzerland
Volkan.Arslan@inf.ethz.ch

ABSTRACT

The **SCOOP** model (Simple Concurrent Object-Oriented Programming) [Mey97] 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 this article, we present **SCOOPLI** for .NET: a library implementation of SCOOP. We focus on the mapping of SCOOP concepts to .NET constructs. We show how *processors* can be mapped to *application domains*, and how *separate calls* are implemented. We also discuss distributed programming with SCOOPLI. Finally, we point out the improvements that may be achieved by the use of .NET, as opposed to the previous, thread-based, implementation.

Keywords
Concurrent object-oriented programming, SCOOP, Eiffel, Design by Contract, distributed programming, .NET Remoting, .NET Threading.

1. INTRODUCTION

The **SCOOP** model has been proposed as a new approach to building concurrent and distributed systems [Mey93] [Mey97]. The basic idea is to take object-oriented programming as given, in a simple and pure form based on the concepts of Design by Contract, which have proved highly successful in improving the quality of sequential programs, and extend them in a minimal way to cover concurrency and distribution. The extension indeed consists of just one keyword ***separate***; the rest of the mechanism largely derives from examining the consequences of the notion of contract in a non-sequential setting.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

1st Int.Workshop on C# and .NET Technologies on Algorithms, Computer Graphics, Visualization, Computer Vision and Distributed Computing

February 6-8, 2003, Plzen, Czech Republic.
Copyright UNION Agency – Science Press
ISBN 80-903100-3-6

The model is applicable to many different physical setups, from multiprocessing to multithreading, network programming, Web services, highly parallel processors for scientific computation, and distributed computation. For application programmers, writing concurrent applications with SCOOP is extremely simple, since it does not require the usual baggage of concurrent and multithreaded programming (semaphores, rendezvous, conditional critical regions etc.). 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.

Although SCOOP has attracted considerable attention, it has only had prototype implementations so far. Our research work is aimed at refining the model and providing a working, production-quality implementation. SCOOP can be implemented in several different environments (Fig. 1) but we have chosen Microsoft .NET to be our reference platform.

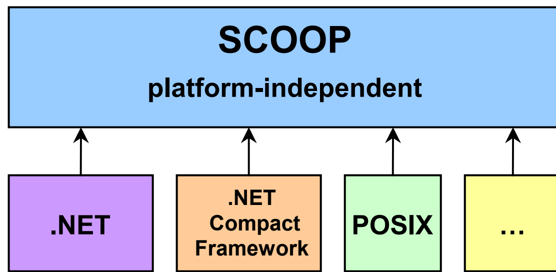

Figure 1. Two-level architecture of SCOOP

.NET offers several mechanisms that seem to be extremely suitable for SCOOP. Most of them are provided by *System.Runtime.Remoting* and *System.Threading* namespaces. In this paper, we present SCOOPLI: a library-based implementation of SCOOP for .NET.

The rest of the article is organised as follows:

Section 2 provides a short description of the SCOOP model. Section 3 presents the SCOOPLI library. Section 4 focuses on mapping of SCOOP concepts to .NET constructs and describes how distributed execution can be achieved. Finally, Section 5 summarises the article and describes our future research directions.

Eiffel notation is used in all code examples.

2. THE SCOOP MODEL

SCOOP stands for *Simple Concurrent Object-Oriented Programming*. Indeed, the very power of the model lies in its simplicity. More precisely, the extension covering full-fledged concurrency and distribution is as minimal as it can get starting from a sequential notation: SCOOP adds a single new keyword to the Eiffel programming language — **separate**.

2.1. Processors

Processors are the principal new concept for adding concurrency to the framework of sequential object-oriented computation. A concurrent system may have any number of processors, as opposed to just one for a sequential system. One has to be very careful when using the word "processor": it should not be confused with a physical CPU! In the SCOOP model, a processor is an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects. It can be implemented by a piece of hardware (a computer), but also by a process of the underlying operating system, or, on multithreaded operating systems, a thread of such a process. In the .NET Framework, processors can be mapped to *application domains* (see section 4). Viewed by the software, processor is an abstract concept; the same concurrent application may be executed on very different architectures (time-sharing on one computer, multiple threads within one Unix or Windows process, etc.) without any change to its source text.

SCOOP uses the fundamental scheme of the O-O computation: feature call, x.f (a), executed on behalf of some object O1 and calling operation f on object O2 attached to x, with the argument a. In a sequential setting, a single processor handles operations on all objects, therefore feature calls are synchronous. This means that the execution of the next feature call will not begin before the current call has terminated.

Concurrency is introduced by allowing the use of multiple processors. What happens if we rely on different processors for handling O1 and O2? The computation on O1 can move ahead without waiting for the call on O2 to terminate, since another processor handles it (Fig. 2). Hence the asynchronous semantics of such feature calls.

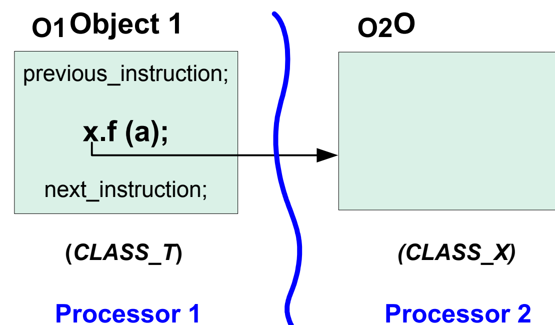

Figure 2. Asynchronous call in SCOOP

2.2. Separate objects

Since the effect of a call depends on whether the caller and the callee objects are handled by the same processor or by different ones, the software text must indicate that fact unambiguously. A declaration of an entity or function, which normally appears as x: SOME_CLASS may now also be of the form x: **separate** SOME_CLASS. Keyword *separate* indicates that entity x is handled by a different processor, so that calls on x should be asynchronous and can proceed in parallel with the rest of computation. With such a declaration, any creation instruction create x.make (…) will spawn off a new processor to handle calls on x. Please note that we do not specify which processor to use for handling. The important thing is the fact that this processor is different from the processor handling the current object.

Instead of declaring a single entity x as separate, the declaration of its base class may also be of a new form: **separate class** SOME_CLASS. In this case SOME_CLASS will be called *separate class*[1]. The following conventions follow:

- a type is separate if:
  - it is based on a separate class, or
  - it is of the form **separate** T for some T (T itself may be non-separate or separate),
- an entity is separate if its type is separate,
- an object is separate if it is attached to a separate entity,
- a function is separate if its type is separate,
- an expression is separate if it is either a separate entity or a call to a separate function,
- a call or creation instruction is separate if its target is a separate expression,
- a precondition clause is separate if it involves a separate call.

2.3. Consistency rules

The validity of separate calls is governed by the *Separateness Consistency Rule* [Mey97]:

- If the source of an attachment (assignment instruction or assignment passing) is separate, its target entity must be separate too.
- If an actual argument of a separate call is of a reference type, the corresponding formal argument must be declared as separate.
- If the source of an attachment is the result of a separate call to a function returning a reference type, the target must be declared as separate.
- If an actual argument of a separate call is of an expanded type, its base class may not include, directly or indirectly, any non-separate attribute of a reference type.

For a separate call to be valid, the target of the call must be a formal argument of the enclosing routine. If an assertion contains a function call, any actual argument of that call must, if separate, be a formal argument of the enclosing routine, if any.

2.4. Access control policy

As mentioned above, the target of a separate call must be a formal argument of the enclosing routine. Such "embedding" of separate calls in routines has one more purpose: it allows exclusive locking of separate objects. In order to obtain exclusive access to a separate object O2, it suffices to use the attached entity (e.g. a) as an argument of the corresponding call, as in r (a).

A routine precondition [Mey97] involving a separate argument causes the client to wait until the precondition holds. Therefore, such a precondition becomes *wait condition* (see 3.3.3).

The access control policy in SCOOP is very restrictive: at any given time, at most one routine can be executed by a processor in charge of the separate object. This restriction ensures that concurrent execution of routines will not break the class invariant.

Sometimes, however, there may be need to *interrupt* the execution of a routine to let a new, high-priority client take over. The concept of *duel* [Mey97] has been introduced to handle such situations.

2.5. Synchronisation

No special mechanism is required for a client to resynchronize with its supplier after a separate call x.f (a) has gone off in parallel. The client will wait if and only if it needs to, i.e. when it requests information on the object through a query call, as in value := x.some_query. This automatic mechanism is known as *wait by necessity* [Car91]. SCOOP ensures that the separate calls made from one client to one supplier are executed in the right (FIFO) order.

3. SCOOPLI

The two-level architecture of SCOOP (Fig. 1) suggests that the general concurrency mechanism (top layer) should be implemented in a platform-independent style. Such concepts as *processor*, *separate object*, and *separate call* are expressed at this level. Only their mapping to platform-dependent constructs will differ from one platform to another. In this section, we describe the top layer, i.e. the implementation of the general concurrency mechanism. 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. There are two main reasons for this choice: first of all, a library-based solution allows for more flexibility in "playing" with the model, i.e. refining and extending it; secondly, it allows us to implement SCOOP on several platforms (e.g. .NET, POSIX threads, etc.) without taking care of very complex compilation-related issues. Nevertheless, the final

[1] It follows from the syntax convention that a class may be at most one of: separate, expanded, deferred. The separateness of a class is not inherited: a class is separate or not according to its own declaration, regardless of its parents' status.

production-quality implementation will be provided as the extension to the Eiffel compiler.

Let's have a look at the functionality provided by the SCOOPLI library, and see how it can be used by programmers.

3.2. Basic concepts

The library 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. [Mey97]

Following this definition, a ***separate client*** is a class which contains a declaration of the form x: **separate** S[2]. S is then said to be a ***separate supplier***.

A separate client is handled by a different processor than each of its separate suppliers[3]. Therefore, any call of a feature on the separate supplier by the separate client (we are going to call it a *separate call*) is executed asynchronously, i.e. the separate client can move to the next instruction without waiting for the current call to terminate.

3.3. Interface

Working with SCOOPLI, one cannot use the exact SCOOP syntax, since SCOOPLI rests on a library-only approach. The criteria that guided the design of the interface were to make it as simple and easy to use as possible, and to maintain a clear correspondence with the SCOOP syntax.

4.2.1 Declaration of a separate supplier

In SCOOP, a declaration of a separate supplier can be expressed as:

a) x: **separate** S

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

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

```
class SEPARATE_S
  inherit
    SEPARATE_SUPPLIER
    S
…
end

x: SEPARATE_S
```

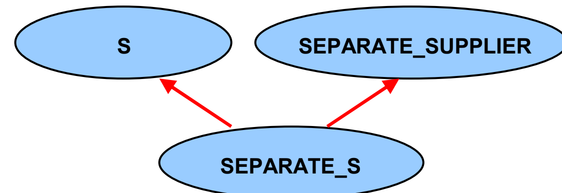

Figure 3. Multiple inheritance allows declaration of separate entities in SCOOPLI

4.2.1 Declaration of 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, multiple inheritance is used: in the same way as separate supplier classes inherited from SEPARATE_SUPPLIER, every separate client class must inherit from SEPARATE_CLIENT.

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

4.2.1 Separate procedure calls

Direct application of features on separate supplier objects is prohibited in SCOOP (see 2.3). This means that we cannot write just x.f(a), if x is separate. We should "embed" the call to x.f(a) in a routine:

```
-- in class MY_CLASS
r (a_x: separate S; a: SOME_CLASS) is
     -- execute a_x.f (a)
   do
     a_x.f (a) -- here, a separate call is allowed
   end

…
r (x, a) -- here, a direct call to x.f (a) is prohibited
```

[2] This is expressed in the SCOOP syntax. The actual syntax of SCOOPLI is slightly different (see 3.1.1).

[3] In fact, SCOOP allows attaching a non-separate 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.

```
            -- we use r (x, a) instead
```

There may be several separate calls to one or more separate suppliers within one routine. All these separate suppliers must be formal arguments of the routine. The locking mechanism of SCOOP is based upon this convention: before executing the routine, the separate client object obtains exclusive locks on all separate supplier objects passed as actual arguments to the routine (see 2.4).

SCOOPLI follows the SCOOP style, with a different syntax:

```
-- in class MY_CLASS
r (a_x: SEPARATE_S; a: SOME_CLASS) is
        -- execute a_x.f (a)
    do
        separate_routine (a_x, agent a_x.f (a))
            -- corresponds to a_x.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's have a closer look at them.

separate_routine (supplier: SEPARATE_ SUPPLIER; procedure: PROCEDURE [])

Formal arguments:

- supplier

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

- procedure

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

In the example above, separate_routine is called with arguments a_x (for supplier) and **agent** a_x.f (a)[4] (for procedure). Such call corresponds 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[5] 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 call corresponds to r (x, a) in SCOOP.

1.3..1 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 a_wait_condition to separate_execute, e.g.

```
r (a_x: SEPARATE_S; a: SOME_CLASS) is
    require
        x_not_empty: not x.is_empty
        a_positive:     a > 0
    do
        separate_routine (a_x, agent a_x.f (a))
            -- corresponds to a_x.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)
```

4.2.1 Separate function calls

Direct application of features on separate supplier objects is prohibited (see 2.3, 3.3.3). 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

---

[4] **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 [ETL3]. One can think of agents as a more sophisticated form of .NET *delegates*.

[5] Wait condition is the part of a routine precondition that involves separate objects.

of x.some_value must be embedded in a routine that takes x as argument.

```
-- in class MY_CLASS
y: T
…
r (a_x: separate S) is
      -- assign a_x.some_value to y
   do
      y := a_x.some_value
   end
…
r (x)
```

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

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

Let's have a closer look at the syntax:

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

Formal arguments:

- supplier

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

- function

  Denotes the function to be evaluated.

Return value is of type ANY.

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

The reason for using the *reverse assignment*[6] (?=) instead of the standard one (:=) is that separate_value always returns an object of type ANY, which must be converted to an object of type T (corresponding to the left-hand side of the assignment).

NB: If the evaluated function returns an object of an *expanded type*[7], a dedicated routine is used instead of separate_value, e.g. separate_boolean_value for BOOLEAN, separate_integer_value for INTEGER, etc. No reverse assignment is needed in such cases.

separate_execute is used in the same way as for separate procedure calls (see 3.3.3).

4. SCOOP ON .NET

In this section we describe how logical processors of the SCOOP model (see 2.1) can be mapped to application domains. We also show how the multithreading model of the Microsoft .NET Framework is used in our implementation.

4.1. Mapping of processors to application domains

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*. Application domains can be considered as managed logical sub-processes. They provide isolation, unloading and security boundaries for managed .NET code. By using several application domains within a process, server scalability can be greatly increased [NET02].

Threads are operating system constructs which execute managed code within an application domain. Therefore, threads can be defined as paths of execution. There is no one to one correlation between threads and application domains, i.e. 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 [Den03], thanks to the remoting capabilities of the .NET Framework.

In the implementation of the SCOOPLI library for the .NET platform processors are mapped to application domains. As you can see in Fig. 4, the processor which handles the objects o1, o2, and o3, is mapped to the application domain 1. The processor

---

[6] Reverse assignment is similar to a cast, with one major difference: if a reverse assignment cannot be made, no exception is raised; the left-hand side of the assignment receives then value Void.

[7] BOOLEAN, INTEGER, REAL, DOUBLE, CHAR, and any other type based on an *expanded class* [Mey97]

which handles the objects o4, o5, and o6, is mapped to the application domain 2, and so on.

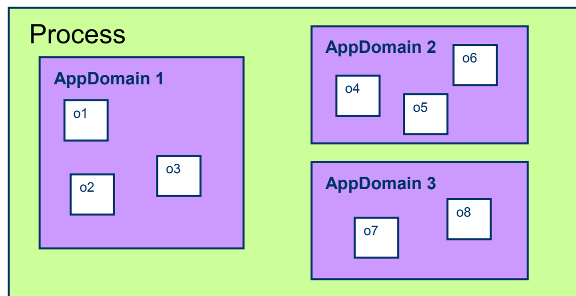

Figure 4. Application domains and separate objects

4.2. Distributed execution

In the SCOOPLI library, distributed execution is made possible by using the concept of application domains of the .NET Framework.

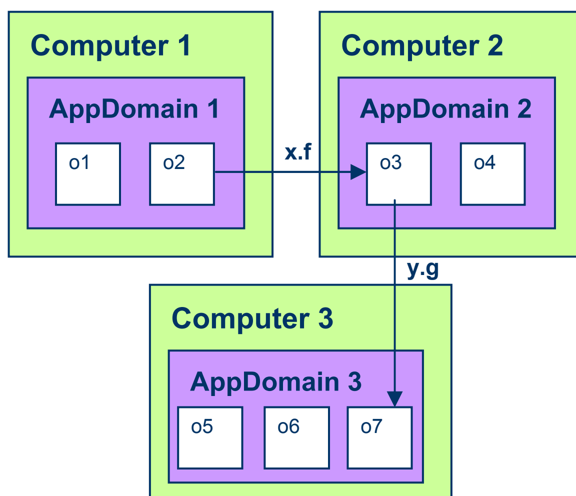

Figure 5. Distributed execution in SCOOPLI for .NET

Let's have a closer look at the example in Fig. 5. 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 AppDomain located on a different computer than o3, call y.g has also separate (asynchronous) semantics. This mechanism makes possible distributed execution with several computers. Since processors are mapped to application domains, they can be located on different machines.

4.3. Mapping of processors to application domains

The mapping of processors to application domains is not specified in the software text. Instead, the *Concurrency Control File* (CCF) is used. CCF specifies the mapping of processors to actual physical resources: application domains, threads, web services, etc. In SCOOPLI for .NET, only application domains are considered.

Here is a typical example for such a CCF (the exact format is not very important):

```
creation
  local_nodes:
    system
      "pushkin" (2): "c:\prog\appl1\appl.exe"
      "akhmatova" (4): "c:\prog\appl2\appl2.dll "
      Current: "c:\prog\appl1\appl1.exe"
    end
  remote_nodes:
    system
      "lermontov": "c:\prog\appl3\appl3.exe"
      "tiuchev" (2): "c:\prog\appl4\appl4.exe"
    end
end
external
      Matisse_handler: "mandelstam" port 9000
      ATM_handler: "pasternak" port 8001
end
default
      port: 8001; instance: 10
end
```

The *creation* part specifies which physical resources should be used for separate creations of the form create x.f, where x is separate. The next two parts, called *local_nodes* and *remote_nodes*, deal with the mapping of processors to AppDomains. 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 *pushkin*,
- the next four separate objects will be created in the application domain *appl2*.dll on the computer *akhmatova*,

- 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 *pushkin*, four on *akhmatova*, and so on.

We can also use AppDomains specified in *remote_nodes* and benefit from computers *lermontov* and *tiuchev* to create separate objects. In the software text, we can choose between both groups by using a feature of the facility class CONCURRENCY [Mey97].

The *external* part specifies which physical resources are used for existing separate objects. In the example above, we can get a reference to a separate database object from the computer *mandelstam* on port 9000 by using an appropriate function server

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

with the argument "Matisse_handler".

The CCF file is separate from the software text. What's more, it is not a compulsory part of a SCOOP-based application. If CCF exists, the mapping of the processors would be done according to the information in the file. Should CCF be not available, the standard mapping scheme is used: every processor is mapped to an application domain on the current computer. The compilation of a concurrent application using SCOOP or the SCOOPLI library is completely independent from the existence or non-existence of a CCF.

5. CURRENT LIMITATIONS AND FUTURE WORK

We have presented SCOOPLI for .NET: a library for concurrent object-oriented programming. We have provided a concise summary of the SCOOP mechanism. The interface of the library has been discussed and compared with the original SCOOP syntax. We have also shown how *processors* can be mapped to *application domains*, and how *separate calls* are implemented. Distributed programming with SCOOPLI has been described. Thanks to the use of .NET Remoting, the implementation of the distributed execution has been greatly simplified, compared to the previous, thread-based version of SCOOPLI.

The following features of SCOOP have been implemented so far:

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

Future developments will include:

- evaluation of functions implemented as attributes,
- argument passing (reference types),
- exclusive locking of several separate objects at a time, with all its implications for wait by necessity, wait conditions, etc.

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". This project has been financially supported by the Hasler Foundation (Berne Switzerland).

We would like to thank Bertrand Meyer for his comments and suggestions.

7. REFERENCES

[Car93] Caromel, D. *Towards a Method of Object-Oriented Concurrent Programming*, in CACM, Communications of the ACM, Volume 36, Number 9, September 1993, pp. 90-102.

[Den03] Dennis, A. *.NET multithreading*, 1st edition, Manning, 2003

[ETL3] Meyer, B. *Eiffel: The language*, 3rd edition, to be published, Prentice Hall

[Mey93] Meyer, B. *Systematic Concurrent Object-Oriented Programming*, in Communications of the ACM, Volume 36, Number 9, September 1993, pp. 56-80.

[Mey97] Meyer, B. *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997

[NET02] *.NET Framework SDK Documentation*, Microsoft, 2002

[Ram02] Rammer I. *Advanced .NET Remoting*, 1st edition, Apress, 2002